IN THE UNITED STATES DISTRICT COURT
FOR THE DISTRICT OF DELAWARE

FINJAN SOFTWARE, LTD., an Israel corporation,	)	
	)	
Plaintiff-Counterdefendants,	)	
	)	
v.	)	Civil Action No. 06-369-GMS
	)	
SECURE COMPUTING CORPORATION, a Delaware corporation; CYBERGUARD CORPORATION, a Delaware corporation, WEBWASHER AG, a German corporation and DOES 1 THROUGH 100,	)	**REDACTED PUBLIC VERSION**
	)	
Defendants-Counterclaimants.	)	

DEFENDANTS-COUNTERCLAIMANTS' OPENING BRIEF IN SUPPORT OF MOTION FOR JUDGMENT AS A MATTER OF LAW PURSUANT TO FED. R. CIV. P. 50 AND, IN THE ALTERNATIVE, MOTION FOR NEW TRIAL AND/OR REMITTITUR PURSUANT TO FED. R. CIV. P. 59

Fredrick L. Cottrell, III (#2555)
cottrell@rlf.com
Kelly E. Farnan (#4395)
farnan@rlf.com
Richards, Layton & Finger
One Rodney Square, P.O. Box 551
Wilmington, DE 19899
(302) 651-7700
Attorneys For Defendants-Counterclaimants

OF COUNSEL:
Ronald J. Schutz
Jake M. Holdreith
Christopher A. Seidl
Trevor J. Foster
Robins, Kaplan, Miller & Ciresi L.L.P.
2800 LaSalle Plaza
800 LaSalle Avenue
Minneapolis, MN 55402

Dated: April 25, 2008

TABLE OF CONTENTS

TABLE OF AUTHORITIES

O2 Micro Int'l Ltd v. Beyond Innovation Tech. Co.,
2008 U.S. App. LEXIS 7053, Fed. Cir. Apr. 3, 2008 .... 11, 13

Pharmastem Therapeutics, Inc. v. Viacell, Inc.,
2004 U.S. Dist. LEXIS 25176 (D. Del. Dec. 14, 2004,
aff'd in part, rev'd in part, 491 F.3d 1342 (Fed. Cir. 2007) 3, 23

ResQNet.com v. Lansa, Inc.,
533 F. Supp. 2d 397 (S.D.N.Y. 2008) 43, 44

Southwest Software, Inc. v. Harlequin, Inc.,
226 F.3d 1280 (Fed. Cir. 2000) 21

Symantec Corp. v. Computer Assocs. Int'l, Inc.,
2008 U.S. App. LEXIS 7826 (Fed. Cir. Apr. 11, 2008) 50

TI Group Auto. Sys., Inc. v. VDO N. Am.,
2002 U.S. Dist. LEXIS 17783 (D. Del. Sept. 4, 2002) 3

TPIG, Inc. v. AT&T Corp.,
2007 U.S. Dist. LEXIS 79919 (E.D. Tex. Oct. 29, 2007) 42

Trading Techs. Int'l, Inc. v. eSpeed, Inc.,
2008 U.S. Dist. LEXIS 295 (N.D. Ill. Jan. 3, 2008) 42, 44, 46

Verizon Servs. Corp. v. Vonage Holdings Corp.,
503 F.3d 1295 (Fed. Cir. 2007) 41

Warner-Jenkinson Co. v. Hilton Davis Chem. Co.,
117 S. Ct. 1040 (1997) 5

Wleklinski v. Targus, Inc.,
2007 U.S. App. LEXIS 29409 (Fed. Cir. Dec. 19, 2007) 10

Statutes

35 U.S.C. § 271(a) 21

Fed. R. Civ. P. 50(a) 2, 48

Fed. R. Civ. P. 50(b) 1, 3, 4, 52

Fed. R. Civ. P. 59 .... 1, 3, 4, 52

INTRODUCTION

The jury's March 12, 2008, verdict against Defendants-counterclaimants Secure Computing Corporation, Cyberguard Corporation, and Webwasher AG ("Secure") should be overturned, pursuant to Fed R. Civ P. 50(b), because no reasonable jury could have arrived at the conclusions this jury did. In the alternative, because the verdict was against the weight of the evidence, the damages awarded are excessive, and there were substantial errors of law, the Court should grant a new trial, or remittitur, pursuant to Fed. R. Civ. P. 59

Specifically, the Court should grant: judgment as a matter of law ("JMOL"), or in the alternative a new trial, on non-infringement, invalidity, and no willful infringement of Finjan's United States Patent Nos 6,092,194 ("the '194 Patent"), 6,804,780 ("the '780 Patent"), and 7,058,822 ("the '822 Patent"); JMOL, or in the alternative remittitur and/or a new trial, on alleged damages relating to alleged infringement of the '194, '780, and '822 Patents; and JMOL, or in the alternative a new trial, on Finjan's infringement of Secure's U S Patent No. 7,185,361 ("the '361 Patent"), and damages caused by such infringement. As explained in this brief, Secure should be granted JMOL, or in the alternative a new trial or remittitur, because *inter alia* the jury's verdict of infringement is contrary to the Court's claim construction and to law, because the asserted patent claims are invalid as a matter of law based on the undisputed facts, because there is insufficient evidence of willful infringement, and because the damages award is excessive and founded on errors of law.

I. NATURE OF THE PROCEEDINGS AND BACKGROUND

This action involves allegations of infringement relating to five United States Patents The relevant facts for this motion are contained in trial transcripts (*See* D I 227-234) (cited to herein as

"(Tr. at page:line (D.I.).") and the trial exhibits from the eight-day trial, as well as the parties' previous *Markman* briefs, and trial-related motions and documents.

Finjan asserts that Secure's Webwasher product and Cyberguard TSP product infringe claims 1-4, 24-30, 32-36 and 65 of the '194 Patent, claims 1-6, 9-14 and 18 of the '780 Patent, and claims 4, 6, 8, 12 and 13 of the '822 Patent. At trial, Secure asserted several substantial defenses, including non-infringement and invalidity of Finjan's Patents. Secure asserts that Finjan's Vital Security products infringe claims 1-5, 7-12 and 14-15 of the '361 Patent."[1]

II. STAGE OF THE PROCEEDINGS

This case was tried to a jury from March 3-11, 2008. (D.I. 227-233.) On March 6, 2008, at the close of Finjan's case-in-chief and pursuant to Fed. R. Civ. P. 50(a), Secure timely moved for JMOL.[2] (Tr. at 687:1-689:16 (D.I. 230); *see also* D.I. 217.) The Court denied Secure's motions, with the exception of Secure's motion for JMOL regarding Finjan's willful infringement claim, on which the Court reserved judgment. (*Id.* at 688:6-689:16 (D.I. 230).) Prior to Finjan's rebuttal case, the Court ultimately denied JMOL on non-willfulness. (*Id.* at 1268:7-1270:1 (D.I. 232).)

On March 12, 2008, at the close of all the evidence, Secure again timely moved for JMOL and renewed its previous motions for JMOL. (D.I. 222.) On March 12, 2008, the jury returned a verdict as follows: (1) Secure literally infringes claims 1-2, 4-14, 24-30, 32-36, and 65 of the '194 Patent; (2) Secure infringes claim 3 of the '194 Patent under the doctrine of equivalents; (3) Secure infringes claims 1-6, 9-14, and 18 of the '780 Patent under the doctrine of equivalents; (4) Secure literally infringes claims 4, 6, 8, and 12-13 of the '822 Patent; (5) Secure's infringement was willful; (6) Secure did not prove that Finjan's patents are invalid; (7) Finjan is entitled to damages

[1] Secure is not briefing the post-trial issues regarding U.S. Patent No. 6,357,010.

[2] In addition, prior to trial, Secure filed: claim construction briefs (D.I. 111, 120); three motions in limine: (D.I. 154, 167, 153, 166, 155, 165); documents required by the Pretrial Order; and made various objections and offers of proof during trial.

in the amount of a 16% royalty rate for $49 million in sales of Webwasher Software, an 8% royalty rate for $3,250,000 sales of Webwasher Hardware Appliances, and an 8% royalty rate for $13,500,000 sales of Cyberguard TSP Hardware Appliances, for a total of $9,180,000; (8) Finjan does not infringe Secure '361 and '010 Patents; (9) Secure's '361 and '010 Patents are valid. (Tr. at 1671:12-1678:25 (D.I. 234).) On March 28, 2008, the Court entered Judgment in favor of Finjan, and against Secure, for monetary damages. (D.I 242.) On March 27, 2008, after the verdict and prior to the entry of judgment, Secure timely moved for JMOL, pursuant to Fed R Civ. P. 50(b), and in the alternative, moved for a new trial and/or remittitur (D.I. 240.) On April 11, 2008, following the Court's entry of judgment on the verdict, Secure timely filed an amended motion under Rule 50, for JMOL, and under Rule 59, for a new trial or to alter or amend the judgment (remittitur) (D.I. 253.)

ARGUMENT

III. LEGAL STANDARDS UNDER FED. R. CIV. P. 50 AND 59

A. Renewed Motion For JMOL

Pursuant to Fed. R. Civ. P. 50, the Court may render JMOL, after the moving party is fully heard on an issue at trial, if "there is no legally sufficient evidentiary basis for a reasonable jury to find for that party on that issue." *ISCO Int'l, Inc v. Conductus, Inc*, 279 F. Supp. 2d 489, 493 (D. Del. 2003); *see also Pharmastem Therapeutics, Inc v. Viacell, Inc*, 2004 U.S. Dist. LEXIS 25176, at *6 (D. Del. Dec 14, 2004), *aff'd in part, rev'd in part*, 491 F.3d 1342 (Fed. Cir. 2007); *TI Group Auto. Sys., Inc v. VDO N. Am*, 2002 U.S. Dist. LEXIS 17783, at *3-4 (D. Del. Sept. 4, 2002). To prevail on JMOL pursuant to Rule 50(b), a party "must show that the jury's findings, presumed or express, are not supported by substantial evidence or, if they were, that the legal conclusion(s) implied [by] the jury's verdict cannot in law be supported by those findings." *ISCO*, 279 F. Supp.

2d at 493. "Substantial evidence" is "such relevant evidence from the record taken as a whole as might be accepted by a reasonable mind as adequate to support the finding under review." *Id* For a JMOL motion pursuant to Rule 50(b), "[t]he appropriate inquiry is whether a reasonable jury, given the facts before it, could have arrived at the conclusion it did." *Id*

B. Motion For New Trial

The Court may grant a new trial pursuant to Fed. R. Civ P. 59 "for any of the reasons for which new trials have heretofore been granted in actions of law in the courts of the United States." *Becton Dickinson & Co. v. Tyco Healthcare Group, LP*, 2006 U.S. Dist. LEXIS 14999, at *6-7 (D Del. Mar 31, 2006). Rule 59 does not detail the grounds on which a new trial may be granted, however, the Third Circuit has recognized the following grounds for a new trial: (1) the verdict was against the weight of the evidence such that a miscarriage of justice would result if the verdict were to stand or where the verdict cries out to be overturned or shocks the conscience; (2) damages are excessive; (3) the trial was unfair; and (4) that substantial errors were made in admission or rejection of evidence or the giving or refusal of instructions. *Donald M. Durkin Contracting, Inc. v. City of Newark*, 2008 U S. Dist. LEXIS 28987, at *4 (D. Del. Apr. 9, 2008).

In determining whether a new trial is appropriate, "the trial judge should consider the overall setting of the trial, the character of the evidence, and the complexity or simplicity of the legal principles which the jury had to apply to the facts." *Becton,* 2006 U S. Dist. LEXIS 14999, at *7. While granting a new trial is left to the discretion of the Court, the Court has greater discretion where the subject matter is complex, such as a complex patent infringement matter like the current case. *Cudone v. Gehret,* 828 F. Supp. 267, 270 (D. Del 1993). Finjan does not dispute that this is a complex patent infringement case. (Tr. at 114:6-8 (D.I. 227); 1591:19-21 (D.I. 233).) Therefore, the complexity of this case gives the Court greater discretion to grant a new trial

C. Remittitur

As an alternative to granting a new trial, the Court may reduce the damages award if the Court deems the award to be "intrinsically excessive in the sense of being greater than the amount a reasonable jury could have awarded, although the surplus cannot be ascribed to a particular quantifiable error." *Boyce v Edis Co*., 224 F. Supp 2d 814, 817-18 (D Del. 2002) Remittitur is appropriate when "a properly instructed jury hearing properly admitted evidence makes an excessive award." *Id*. at 818.

IV. THE COURT SHOULD GRANT JMOL, OR IN THE ALTERNATIVE A NEW TRIAL, ON NON-INFRINGEMENT OF FINJAN'S PATENTS

A. No Infringement Under the Doctrine of Equivalents ('780 and '194 Patents)

1. Prosecution History Estoppel is a Bar to Judgment of Infringement of the '780 Patent Under the Doctrine of Equivalents.

Prosecution history estoppel is presumed to apply as a bar to infringement under the doctrine of equivalents if there was—as here—a narrowing amendment to the claim that was made for a substantial reason related to patentability. *Warner-Jenkinson Co v. Hilton Davis Chem Co*., 117 S. Ct 1040, 1051 (1997); *Honeywell Int'l, Inc v. Hamilton Sundstrand Corp*, 2008 U S. App LEXIS 8405 (Fed. Cir Apr 18, 2008)(affirming the district court's decision that prosecution history estoppel bars a doctrine of equivalents analysis)

Prosecution history estoppel is presumed to apply against claims 1-6, 9-14, and 18 of the '780 Patent based on an amendment made to these claims during prosecution, therefore, these claims cannot be infringed under the doctrine of equivalents[3]

[3] These arguments were previously detailed this argument in Secure's motion in limine on the doctrine of equivalents (*See* D I 155, 165.) The Court determined that it was not appropriate to address doctrine of equivalents at that time (Oral Order 2/4/08; Pretrial Tr 39-41 (D.I. 206).)

The amendments that bar Finjan from arguing infringement under the doctrine of equivalents are identified below. For example, claim 1 was amended as follows:

> 1. A computer-based method for generating a Downloadable ID to identify a Downloadable, comprising ~~the steps of~~:
>
> obtaining a Downloadable that includes one or more references to software components required to be executed by the Downloadable;
>
> fetching ~~if the Downloadable includes one or more references to a component~~ at least one software component identified by the one or more references;
>
> and performing a hashing function on the Downloadable and ~~all~~ the fetched software components ~~fetched~~ to generate a Downloadable ID.

See Ex 1, at 3-5; Ex. 2, at 2-4. Independent claims 9 and 18 were also amended with the addition of the same limitations. *Id.*

These amendments plainly narrow the scope of the claims by adding limitations. Finjan made these amendments for substantial reasons of patentability. In particular, the inventors made these amendments in order to avoid rejection based on the prior art references of Apperson et al and Khare ("Authenticode"). Ex 1, at 6. In fact, the inventors admitted that these amendments were made to make their claim patentable. After amending the claims the inventors stated, "[i]n distinction to the present invention, Apperson and Khare do not teach fetching software components of executable code. In order to further clarify this distinction, applicant has amended the claims so as to refer to software components required by the Downloadable." (*Id.* at 7; *see also* Ex. 2, at 5-6 (explaining that the prior art, Apperson and Khare, do not meet the amended limitation that was added to every independent claim)). On a later occasion, the inventors amended to add the limitation "to be executed" and stated "Applicant and the Examiner discussed adding the language 'to be executed' into the claim language to further show that the additional components are 'to be executed' thereby highlighting the difference between the Apperson reference and the claimed invention." Ex. 2, at 5. Both of these amended limitations are present in each of the asserted claims of the '780 Patent. Consequently, prosecution history estoppel is presumed to apply to all of the asserted claims of the '780 Patent.

Finjan did not rebut the overwhelming evidence that prosecution history estoppel applies because the amendment was made to overcome a prior art rejection. Finjan did not meet its burden of rebutting this presumption, in light of the specific evidence that it narrowed the scope of its claims. Thus, prosecution history estoppel applies and, as a matter of law, the jury should not have been allowed to consider the doctrine of equivalents. The Court should grant JMOL to set aside the verdict of '780 equivalent infringement is proper, or order a new trial.

2. Finjan Failed to Present the Legally Required Linking Arguments Sufficient to Prove Infringement of Claim 3 of the '194 Patent and Claims 1-6, 9-14, and 18 of the '780 under the Doctrine of Equivalents.

In addition to the bar, Finjan simply failed to offer sufficient evidence under the doctrine of equivalents, even if there were no estoppel. The Federal Circuit requires that a patentee, in order to prove infringement under the doctrine of equivalents, must present "evidence and argument concerning the doctrine and each of its elements." *nCube Corp. v. SeaChange Int'l, Inc.*, 436 F.3d 1317, 1325 (Fed. Cir. 2006) (quoting *Lear Siegler, Inc. v. Sealy Mattress Co. of Mich., Inc.*, 873 F.2d 1422, 1425 (Fed. Cir. 1989)). The doctrine of equivalents analysis contains three separate elements, "substantial identity of function, means, and result." *Lear*, 873 F.2d at 1425. The Federal Circuit thus requires evidence in the form of "particularized testimony" and linking arguments that explain how the testimony shows substantial identity between the function, means, and result of the alleged equivalent and the claim limitation. *Id.*; *see also MKS Instruments, Inc. v. Advanced Energy Indus.*, 325 F. Supp. 2d 471, 474 (D. Del. 2004). As the Federal Circuit concluded, "[a]bsent the proper . . . showing of *how* plaintiff compares the function, means, and result of its claimed invention with those of the accused device, a jury is more or less put to sea without guiding charts when called upon to determine infringement under the doctrine." *Lear*, 873 F 2d at 1425-26 (emphasis added).

Finjan's technical expert, Dr. Giovanni Vigna, relied merely on his testimony regarding literal infringement As Dr. Vigna admitted: "Q: When you are giving these doctrine of equivalents opinions, are you basing these opinions on the same material that you base your opinion on literal infringement on? A Yes. It was that easy " (Tr. at 430:22-431:1 (D.I 228)) But it is not that easy It is legally insufficient to rely simply on evidence of literal infringement to demonstrate infringement under the doctrine of equivalents. *Lear*, 873 F.2d at 1425

The only times in which Dr Vigna even touched on equivalents, he did not attempt to explain any differences between the accused product and the claim, let alone *how* the accused product performs the substantially equivalent functions, ways, and results of the claims Dr Vigna simply replied, "yes" to conclusory questions (Tr. at 424:23-425:1; 430:5-8; 435:17-20) As the following chart indicates, this is insufficient as a matter of law.

Held Insufficient By Federal Circuit	Testimony Adduced by Finjan
Q. So, as to each claim element that you have analyzed with respect to all the Mustek devices, do they perform the same function as the claim element? A. Yes Q. And do they do it to achieve the same result? A. Yes. Q. And do they do it in the same way? A Yes. *Hewlett-Packard Co v Mustek Sys*, 340 F 3d 1314, 1323 (Fed Cir 2003) ("This testimony falls far short of the long-standing evidentiary requirements for proof of infringement under	Q. Dr. Vigna, going back to the doctrine of equivalents, at the very least, does the Webwasher product perform substantially the same way as the method in Claim 1? A. Yes Q. At the very least, does the Webwasher product yield the same results as the results of the method of Claim 1? A. Yes. Q. That's for each and every element of Claim 1 for each of those analyses? A Correct (*Id* at 368:18-369:2)

the doctrine of equivalents")	

Dr. Vigna gave no testimony at all regarding claim 3 of the '194 Patent or claims 2-6 and 10-14 of the '780 Patent and the doctrine of equivalents. He simply wasn't asked, and there is no evidence in the record at all as to the alleged equivalent infringement of these claims.

Finally, Finjan's counsel never made an argument in closing that purported to link any particularized testimony to the alleged equivalent. He did not point out anything in Webwasher that he argued to be equivalent, he never argued what the differences are between Webwasher and the claim, and he never made an argument that the differences are insubstantial. He merely made a conclusory statement that Webwasher is, at the very least, equivalent (Tr at 1607:9-1607:18 (D I 233).) It is Finjan's burden to come forth with particularized testimony and an affirmative argument showing equivalents, and there was none on which the jury could rely

Thus, the Court should grant JMOL of non-infringement of claim 3 of the '194 Patent and claims 1-6, 9-14, and 18 of the '780 Patent under the doctrine of equivalents, or a new trial.

3. Even if Finjan Could Apply the Doctrine of Equivalents, Finjan's Expert Improperly Applied a Claim Construction Contrary to the Court's Order.

Even if Finjan is allowed to assert doctrine of equivalents by improperly using the same testimony and arguments it used to prove literal infringement, the Court should grant Secure JMOL, because Dr. Vigna applied a claim construction that contradicted the Court's claim construction. The Court construed the limitation of "performing a hashing function on the Downloadable and the fetched software components to generate a Downloadable ID" as "performing a hashing function on the Downloadable *together with its* fetched software components to generate a Downloadable ID."[4] Markman Order. at 2 (D.I. 142) (emphasis added)

[4] This construction was, as the Court noted, consistent with "how the inventor understood and used the term, as evinced by the patent's prosecution history." *Id* The Court relied on the patentee's statements during prosecution in reaching this construction. *Id.*

Dr. Vigna admitted that Webwasher hashes the Downloadable separately and the component separately. (Tr. at 494:22-496:6 (D.I. 229).)[5] If Dr. Vigna's literal infringement analysis is improperly used to justify a finding of infringement under the doctrine of equivalents, then the Court should grant JMOL of noninfringement of the asserted claims of the '780 patent because Finjan is required to show equivalence between the accused product and the Court's claim construction. Dr Vigna's only comparison between Webwasher and the claims was based on his construction that contradicted the Court's construction. Dr. Vigna's construction was that "performing a hashing function together with its fetched software components" somehow means that the general *time* at which a Downloadable is hashed is close in time and proximity to the separate hashing of a software component (Tr. at 489:4 ("together like in time"); *Id* at 489:19-21 ("The fact they are together specially or in time is what means I operate on them as a group together.") This interpretation cannot be reconciled with the Court's construction that the claim requires "performing a hashing function on the Downloadable *together with its* fetched software components to generate a Downloadable ID " Markman Order at 2 (D.I. 142). This is especially true given the Court's reliance on the statements made in the prosecution history to explain the meaning of this limitation. (*Id*.)

If the Court believes that Dr Vigna's testimony was consistent with the Court's construction, then Secure respectfully submits that the construction was in error The claims and file history clearly show that a Downloadable must be combined with the fetched software components and a hashing function must be performed on this combined block of data *See*

[5] The Federal Circuit's holding in *Wleklinski v Targus, Inc.*, 2007 U.S. App LEXIS 29409 (Fed. Cir. Dec. 19, 2007) is instructive. It holds that a strap made of one material is the opposite, not the equivalent, of a strap made from two materials. *Id.* at *9-10 Similarly, hashing strings separately is the opposite not the equivalent of hashing together. *Id.*

Secure's Open. Markman Br. at 29-31 (D.I. 111); Secure's Markman Resp. Br. at 25-26 (D.I. 120). There is nothing to support Dr. Vigna's interpretation that together means together in time. Accordingly, any verdict of infringement of the '780 patent is contrary to this Court's construction and unsupported.

B. No Literal Infringement ('194 Patent)

1. No Infringement of "Addressed to a Client" Limitation

a. The Court Should Grant JMOL Or A New Trial Regarding Noninfringement of the '194 Patent As a Matter of Law Because the Court Did Not Resolve a Dispute Over the Ordinary Meaning of the Limitation "Addressed to a Client."

Secure was prevented at trial from addressing Finjan's construction of the ordinary meaning of "addressed to a client." If the Court finds that Finjan offered evidence that Webwasher receives a downloadable that is "addressed to a client" from which the jury could find infringement, Secure requests an express construction of the term, and a JMOL of noninfringement or a new trial that will give Secure's expert a fair opportunity to rebut the evidence offered by Finjan under such a construction. The Court should have construed "addressed to a client" using a specific definition, and should have permitted Secure's expert to explain the meaning of the term.

The Court's reliance on the plain and ordinary meaning despite the existence of a factual dispute on that matter, and then refusal to allow Secure's expert to testify as to his understanding of the ordinary meaning, was legal error. The Federal Circuit, in *O 2 Micro Int'l Ltd. v. Beyond Innovation Tech. Co.*, held that it was inadequate for the district court to "determin[e] that a claim term 'needs no construction' or has the 'plain and ordinary meaning' . when reliance on a term's 'ordinary' meaning does not resolve the parties' dispute." *O2 Micro*, 2008 U.S. App. LEXIS 7053, at *22 (Fed. Cir. Apr. 3, 2008). In this case, the parties had a dispute over the ordinary meaning of "addressed to a client." Counsel for Secure specifically asked for an express construction to avoid the very problem created by Finjan's gloss on the term. Counsel stated:

> This is one of those terms where we both agree it ought to have its ordinary meaning. We disagree on what the ordinary meaning is. So the reason for the construction here is I am concerned there is going to be a battle of the experts at trial construing the ordinary meaning. So we want to raise now what the ordinary meaning is, so that we don't have their expert saying, the ordinary meaning is if I send it in a letter and it gets to Mr. Andre, it's addressed to a client and my guy saying, no, no, no, the ordinary meaning is it has to have the network address. That is why we are seeking a construction

(Markman Tr. at 35:4-35:15 (D.I. 132).)

Even though a substantial dispute existed over the "ordinary meaning" of "addressed to a client," the Court construed the term to have "its plain and ordinary meaning " Markman Order at 2 (D.I. 142). The Court determined that Secure's construction of "containing the client computer's network address" was too narrow, however the Court never stated what the proper construction was or what was too narrow about that definition of the ordinary meaning *Id* at 2 n.1.

Notwithstanding the remaining dispute as to ordinary meaning, the Court then prohibited Secure's expert, Dr Wallach, from testifying as to his understanding of the ordinary meaning of the phrase at trial. (Tr. at 808:13-810:19 (D I. 230)); *see also* Secure's Offer of Proof (D.I 216). This decision and the Court's evidentiary rulings at trial took away Dr Wallach's ability to comment on the meaning of "addressed to a client" that was offered by Finjan *Id* He could not use his understanding of the ordinary meaning because the Court would not allow him. On the other hand, he could not apply the Court's understanding of the ordinary meaning, because the Court did not give it. Thus, Secure's expert was left to stand idly by while Finjan's experts were allowed to define the term without rebuttal

In the situation—as is the case here—where there exists a dispute as to the "ordinary meaning" and the Court decides to expressly reject one expert's understanding, then it is necessary for the Court to provide the parties with its own understanding of the terms so that the expert may apply that legal conclusion to the claims. *See O2 Micro*, 2008 U.S. App. LEXIS 7053, at *23. Secure urges this Court to reconsider its claim construction and provide an express definition of "addressed to a client." Secure then requests a new trial on infringement of the '194 patent consistent with the Court's construction Of course, the Court need not address this issue if it determines that Finjan did not offer proof that Webwasher receives an incoming Downloadable addressed to a client under Dr. Vigna's interpretation of "addressed to a client" as discussed below.

b. Finjan Adduced Insufficient Evidence To Prove By A Preponderance Of Evidence That The Webwasher Product Receives Downloadables "Addressed To A Client."

No matter what the construction of "addressed to a client," Finjan failed to point out any specific structure or method in Webwasher to fulfill the limitation that the server receives a Downloadable "addressed to a client" as is required by every asserted claim. The Court did not construe the term "addressed to a client." Dr. Vigna explained that in his opinion "addressed to a client" means that the ultimate destination of the information is described as the client (Tr. at 339:3-339:8 (D.I. 230).) Dr. Vigna clarified this definition by using an analogy to passing notes in school. Dr. Vigna explained that if you wanted to pass a note to Jim, it would be "addressed" to Jim under Dr Vigna's definition so long as "you give it to your next-door person and say, Hey, give it to Jim." (Tr. at 339:15-16; 547:4-9 (D.I. 230, 231).) Even under Finjan's construction, therefore, the message must come with an instruction that it should be passed to some further recipient, to be "addressed to a client."

Finjan adduced no evidence that an incoming Downloadable is ever received by Webwasher with the ultimate destination described as the client. Indeed, Finjan offered no evidence whatsoever as to any information or method by which Finjan alleged that a message received by Webwasher is "addressed to the client," and there was a total failure of proof of this element. Finjan's contention was that the message just gets there somehow. Using Dr. Vigna's own analogy, however, there is no evidence that someone passes the Webwasher product a note and says "Hey, give it to Jim." On the contrary, the evidence showed that the Webwasher product does not receive instructions to give the Downloadable to Jim because the client's identity, Jim, is deliberately kept secret by Webwasher. (Tr. at 817:7-818:3 (D.I. 230).) The undisputed evidence was that when a server on the internet passes a message to Webwasher, the server passing the message does not and cannot know that there is any other client behind Webwasher because it lacks the information that the message will be passed anywhere by Webwasher. (*Id.*) The internet server cannot and does not say "pass this message" to anyone, because it thinks Webwasher is the final destination, rather than some further client. (*Id.*) Consequently, Finjan failed to prove that Secure infringes any of the asserted claims of the '194 Patent by receiving a downloadable "addressed to a client."

2. <u>No Infringement of "List of Suspicious Computer Operations" Limitation</u>

a. As A Matter Of Law, Webwasher Does Not Contain A Profile Of A Downloadable That Includes A "List Of Suspicious Computer Operations" As Required By A Proper Construction Of Claims 1-14, 24-30, 32-36, And 65 Of The '194 Patent.

It is clear that the ordinary meaning of a "list of suspicious computer operations" is not synonymous with recording "categories of behavior" as Finjan's witnesses suggested during trial. (Tr. 1601:19-21 (D.I. 233).) Plain English dictates that a "list of certain things" is different from a "list of categories of things." For example, a list of presidential candidates may include Hillary Clinton, Barack Obama, and John McCain. A list of categories of presidential candidates, on the other hand, may include Republicans, Democrats, and Independents. These are fundamentally different lists. Consequently, this Court should hold that a "list of suspicious computer operations" must consist of a list of the operations within a Downloadable as opposed to the categories of behavior to which the computer operations may be associated. And based on this proper construction of the scope of the claims, the Court should grant JMOL of non-infringement on all of the asserted claims of the '194 patent, or a new trial.

As the Federal Circuit held in the seminal *Markman* case, claim construction is a matter of law. *Markman v Westview Instruments, Inc*, 52 F.3d 967, 988-989 (Fed. Cir. 1995) In *Markman*, the Federal Circuit indicated that the district court had a duty to define the proper scope of the claims when the scope is in disputed based on the testimony of witnesses and apparent understanding by the jury at trial. *Id.* at 985. In that case, the patentee argued through witnesses at trial that the term "inventory" was broad enough to include "transaction totals or dollars." *Id.* at 973. The district court did not provide the jury with a construction of the term inventory. *Id.* at 973. The jury returned a verdict of infringement, but the district court granted JMOL of noninfringement because it determined that "inventory," as a matter of law, was limited to "articles of clothing." *Id.* The Federal Circuit affirmed the district court's post-trial construction of "inventory" and ruling of JMOL of noninfringement, because it was the court's duty to construe the term, and the scope of the term was not as broad as argued and apparently understood by the jury.

Much like the *Markman* case, both parties agree on the fact that Webwasher does not parse a Downloadable, identify the particular functions in the Downloadable, and then copy those functions onto a list. (Tr. at 533:6-534:6 (D.I. 229) (Finjan's expert agreeing that Webwasher does not copy the functions in a Downloadable into a list); Tr. 535:14-536:1; 792:10-16 (D.I. 229, 230) (Secure's expert indicating that Webwasher does not copy the computer operations in a Downloadable onto a list).) This is not a factual dispute. The only question to be resolved is whether a "list of suspicious computer operations," as a legal matter of claim construction, requires the list to actually contain the computer operations within the Downloadable Secure submits that the plain answer is yes based on the ordinary meaning of the term and as further supported by the patent specification.

The jury clearly applied a claim construction under which the "list of suspicious computer operations" was interpreted to be synonymous with the "categories of behaviour" used by Webwasher The Court should rule that "list of suspicious computer operations" is not synonymous with categories of behavior as a matter of law, and grant JMOL of non-infringement of all of the asserted'194 Patent claims, or a new trial.

Finjan must prove that Webwasher contains each limitation of the claim exactly, without deviation, in order for a jury to find literal infringement *Litton Sys v Honeywell, Inc*, 140 F.3d 1449, 1454 (Fed. Cir. 1998) ("Literal infringement requires that the accused device contain each limitation of the claim exactly; any deviation from the claim precludes a finding of literal infringement."); *see also General Am Transp Corp. v Cryo-Trans, Inc*, 93 F.3d 766, 770 (Fed. Cir 1996) ("Where a claim does not read on an accused device exactly, there can be no literal infringement ") (quoting *Johnston v. IVAC Corp.*, 885 F.2d 1574, 1580 (Fed. Cir. 1989)).

Dr. Vigna's testimony demonstrates that he was not applying the legal test for literal infringement to the limitation of "Downloadable security profile data includes a list of suspicious computer operations." Instead, he applied a doctrine of equivalents analysis (Tr 349:8-15; 351:18 (D.I. 228).) In fact, Dr. Vigna testified that categories of behavior were not exactly the same thing as a list of suspicious computer operations. For example, Dr Vigna specifically admitted that the profile of behaviors created in Webwasher was not exactly the same as the limitation in the '194 patent claims:

> Q. Is a behavior file, is that equivalent to what is referred to here as a -- a "behavior profile" equivalent to what is referred to here as a "security profile"?
>
> A. Yes. In the patent, I think it's referred to as the "downloadable security profile" or "DSP." That's what I am talking about.
>
> Q. It is exactly the same thing?
>
> A. It is pretty much the same thing.

(*Id.* at 349:8-15.) This testimony shows that even Dr. Vigna recognized that the so-called behavior profile was not exactly the same as the claim limitation of a Downloadable security profile that includes a list of suspicious computer operations. Exact correlation, however, is legally required to show literal infringement as a matter of law. As a result, Secure requests that the Court grant JMOL that Secure does not literally infringe the asserted claims of the '194 Patent, or in the alternative a new trial.

3. If the Court Finds That the Asserted Claims of the '194 Patent Are Not Literally Infringed, Then the Court Should Find JMOL of Noninfringement, Because Finjan Cannot and Did Not Demonstrate Infringement Under the Doctrine of Equivalents.

Finjan cannot succeed as a matter of law on the theory of doctrine of equivalents for the claims of the '194 patent because Finjan is barred based on prosecution history estoppel and Finjan failed to make the legally required linking argument. First, as explained in Secure's previous motion in limine, Finjan made several narrowing amendments substantially related to patentability in the independent claims of the '194 Patent. (D.I. 155, 165) These amendments were made expressly to avoid prior art and thus foreseeable and relevant equivalents. Finjan cannot rebut this presumption of prosecution history estoppel. Moreover, Dr. Vigna again failed to provide any particularized testimony explaining how the Webwasher products contain substantial identity to the claim in function, means, and result. Again, Dr. Vigna merely affirms his opinion that those functions, means, and result are substantially the same without offering any explanation. And again, Dr. Vigna did not make any attempt, even a conclusory one, to give an opinion on the function means, and result test with respect to any of the dependent claims of the '194 Patent Thus, if the Court finds that there can be no literal infringement as a matter of law, the Court should order that the claims of the '194 Patent are not infringed as a matter of law, or grant a new trial

C. **The Court Should Grant JMOL Or A New Trial, On Noninfringement Of The '822 Patent, Because Finjan Applied Webwasher To A Legally Incorrect Construction Of The Claims**

The '822 Patent claims the execution of specific "if-then" logic in order to ensure that mobile protection code is always sent to a client if the program detects executable code This is based on the plain and ordinary meaning of the claim terms, and is supported by the specification and file history Finjan's expert applied a different construction that does not comport with the plain and ordinary meaning of the claims in light of the intrinsic evidence, and thus must be rejected as a matter of law.

The limitation that claims the specific "if-then" logic reads as follows: "causing mobile protection code to be communicated to at least one information-destination of the downloadable-information, if the downloadable-information is determined to include executable code" In this case, the limitation is written in an inverted "if-then" form, so the claimed "if-then" logic is properly read as "if executable code is detected then communicate mobile protection code (i e. sandbox)." So, in order to meet this limitation, the product actually must execute this particular logic If a product detects executable code but does not send mobile protection code, then that proves that the product does not execute the logical statement of "if executable code is detected then communicate mobile protection code (i e. sandbox)" This is necessarily the case, because if the product executed the claimed "if-then" logic, then *every time* it detected executable code, mobile protection code would be communicated to the client As Secure's unrebutted expert testimony indicates, this understanding of an "if-then" clause is especially true in the context of computer program claims to be interpreted by a computer programmer. (Tr. at 1061:22-1062-18; 1098:17-1099:3 (D.I. 231).) In fact, Dr. Vigna admitted that this "if-then" clause defined a "condition" and said that it means "you send [mobile protection code] if the downloadable contains code, yes." (Tr. at 1561:2-1561:10 (D.I. 229).)

Finjan is accusing different "if-then" logic then is claimed. Finjan has accused Webwasher of infringing this limitation because it performs the following two "if-then" logical statements, "if VBscript is detected then communicate mobile protection code" and "if JavaScript is detected then communicate mobile protection code." These two logical statements are different than the claimed "if-then" logic. And that difference is extremely important, as Finjan made clear in the intrinsic record.

Finjan included this limitation in order to differentiate its invention to prior art inventions that communicated mobile protection code only if it detected a specific type of executable code, for example ActiveX. Specifically, the inventors criticized the prior art of Shuang because it "teaches a protection system for protecting against *only* distributable components including 'Java applets or ActiveX controls.'" '822 Patent, Col.1 ll.64-66 (emphasis added). Likewise, the inventors criticized another prior art reference, Golan, for "focus[ing] *only* on protecting against ActiveX controls *and not other distributable components, let alone other Downloadable types*." *Id.* at col.2, ll.5-7 (emphasis added). These criticisms explain why Finjan claimed the specific step of executing the logic "if executable code is detected then communicate mobile protection code (i.e. sandbox)." Namely, the prior art already communicated mobile protection code based on the detection of specific types of executable code (e.g., ActiveX or Java Applets), but not all types of executable code The patent claims were written in this manner to cover a system that logically detected and protected against *all* types of executable code.

Dr Wallach's testimony indicating that Webwasher does not send mobile protection code based on detection of executable code other than javascript and VBscript is unrebutted (Tr at 987:23-989:22 (D I. 231).) In fact, Dr. Vigna admits that when Webwasher identifies executable code such as ActiveX and Java Applets, it does not sandbox the code. (Tr at 564:8-564:23 (D I 229).) Because it was legally incorrect for Finjan to argue that performing the steps of communicating mobile protection code in response to detecting a specific type of executable code meets the claim limitation, this Court should grant Secure JMOL of noninfringement, or in the alternative, a new trial, on the asserted claims of the '822 Patent.

D. Finjan Failed To Present Evidence That Any Method Claim Is Infringed

Claims 1-14, and 24-30 of the '194 Patent, claims 1-6 of the '780 Patent, and claims 4, 6, and 8 of the '822 Patent are method claims. As a matter of law, a patentee can prove that a method claim of a patent is directly infringed only by showing that someone performed the patented method. *See Joy Tech., Inc. v. Flakt, Inc*, 6 F.3d 770, 773 (Fed. Cir 1993) (stating that a method claim "is directly infringed only when the process is performed"); *see also BMC Resources, Inc. v. Paymentech, L P*, 498 F.3d 1373, 1378 (Fed. Cir. 2007) ("For process patent or method patent claims, infringement occurs when a party performs all of the steps of the process ") Finjan presented no evidence that Secure ever performed the methods at issue. Finjan presented no evidence of anyone performing these methods. Further, Finjan presented no evidence of inducement, and no evidence of intent to induce. Indeed, Finjan abandoned its inducement charge and cannot rely on inducement to infringe as a matter of law (*See* Ex. 3 (email stating that Finjan will not assert indirect infringement at trial); D.I. 225 (no indirect infringement instruction for

Finjan); D.I. 220 (no indirect infringement verdict question for Finjan)). Finjan asserted only direct infringement under 35 U.S.C. § 271(a). Failure to prove that a defendant performed a method claim is a total failure of proof of infringement of that claim *See Crystal Semiconductor Corp. v TriTech Microelectronics Intern., Inc.*, 246 F.3d 1336, 1351 (Fed Cir 2001) (stating that, since defendant did not practice the claimed method, it could not be liable for direct infringement of a method claim); *E I Dupont De Nemours & Co v Monsanto Co*, 903 F Supp. 680, 734 (D. Del. 1995) (holding that defendant who only made and sold a product, but did not actually perform the method, was not liable for direct infringement) Finjan's failure of proof is fatal to Finjan's infringement claims for claims 1-14, and 24-30 of the '194 Patent, claims 1-6 of the '780 Patent, and claims 4, 6, and 8 of the '822 Patent and justifies JMOL or a new trial.[6]

E. <u>As A Matter of Law, Accused Products On Which Alleged Infringing Technology Is Locked, Not Enabled, And/Or Otherwise Not Available, Do Not Infringe Any of Finjan's Patents</u>

The Court erred in refusing to instruct the jury regarding Secure's Proposed Jury Instruction No. 19.2 – Infringement – Functionalities, therefore, the Court should, at the very least, grant a new trial. As a matter of law, software and appliances that have an allegedly infringing functionality in the software source code do not infringe a patent if the user cannot access or use the accused functionality, or if the accused functionality is locked or not enabled *Southwest Software, Inc v Harlequin, Inc.*, 226 F.3d 1280, 1291 (Fed. Cir. 2000); *ISCO*, 279 F Supp. 2d at 506-07. Secure's Proposed Instruction No. 19.2 would have properly instructed the jury on this

[6] Further, Secure should at least be granted a new trial because it was substantial error for the Court to refuse to instruct the jury on method claim infringement (*See* D.I 218, p 24 (Secure's proposed method claim infringement instruction); Tr. at 1410:1-1412:19 (Court overruling Secure's proposed instruction)(D I. 232). Without this instruction, the jury could not have even known how to determine whether, as described above, Finjan failed to prove method claim infringement. If no JMOL is granted, then at least a new trial is warranted based on jury instruction error.

law. (*See* D.I. 218, p. 28.) Without Proposed Instruction No. 19.2, the jury was unable to appropriately determine Finjan's infringement claims, and a new trial should be granted. But a new trial can be avoided by granting JMOL to Secure, because Secure presented substantial unrebutted evidence of at least $24,376,402 in sales[7] of Webwasher software and $12,131,199 in sales of Cyberguard TSP appliances (or all alleged Cyberguard TSP sales) where the accused functionalities is unavailable to customers, or is locked or not enabled. (Tr. at 702:1-706:19; 711:1-714:6 (D.I. 230); 1154:5-20;1115:1-1116:11 (D.I. 231); Exs. 4, 5 (Degen Figures 2 and 2A).) A reasonable and properly instructed jury would have found that these sales do not infringe. Therefore, the Court should grant JMOL that these sales are not infringing, or in the alternative a new trial or remittitur

V. THE COURT SHOULD GRANT JMOL, OR IN THE ALTERNATIVE A NEW TRIAL, ON INVALIDITY OF FINJAN'S PATENTS

A. The '194 Patent Is Invalid

1. No Reasonable Juror Could Conclude that Claims 1-14, 24-30, 32-36, and 65 of the '194 Patent are Valid Based on the Court's Claim Construction.

Finjan's '194 patent describes an antivirus technique that uses a network "gateway" to analyze suspicious computer operations and determine whether to block or allow those operations Based on the Court's claim construction, there can be no dispute that the prior art disclosed this technique before Finjan filed its patent. One patent, the Shaio patent, U.S. Pat. No. 6,571,338 (Ex. 6)[8], directly discloses this technique. Two patents, Ji 1995, U.S. Pat. No. 5,623,600 (Ex. 7) and Chen, U.S. Pat. No. 5,951,698 (Ex. 8), disclose this technique in combination. As the Federal Circuit has made clear, the district court should order JMOL in favor of invalidity, even if it is

[7] The amount of locked sales may be larger, since this number reflects the sales after Mr Degen deducted non-U.S. sales. If non-U.S. sales are not deducted by the Court, those sales are subject to the "locked sales" deduction.

[8] Also incorporating by reference U.S. Pat. No. 5,740,441 (Ex. 9).

contrary to a jury's finding of validity, when "it was unreasonable for the jury to reach the opposite conclusion." *Pharmastem*, 491 F.3d at 1367 It was unreasonable, in this case, for the jury to reach the conclusion that the '194 patent is valid.

a. The Shaio Patent (U.S. Pat. No 6,571,338) Anticipates Claims 1, 32, and 65 of the '194 Patent

The Shaio reference anticipates the claims as a matter of law Both experts agree that Shaio describes a firewall that intercepts packets of information at the border of a company's network (Tr. at 1279:20-1280:1 (describing "firewalls"); 1357:22-1358:2 (referring to his earlier description) (D.I. 232).) Both experts agree that Shaio describes analyzing network packets for java applet instructions (i.e. bytecode) in order to determine whether to block those packets (*Id.* at 1357:27-1358:2.) The only dispute between Finjan and Secure is on the proper scope of the '194 claims. Finjan's expert convinced the jury that the claims narrowly require a "gateway" that receives multiple packets, reassembles them, and then analyzes the code The jury's reliance on this application of an improperly narrow construction of the claim scope is both incorrect as a matter of law, and contrary to the Court's claim construction. Consequently, this Court should grant Secure JMOL of invalidity against independent claims 1, 32, and 65, or at the very least, Secure should be granted a new trial on the issue of invalidity of the '194 Patent

During trial, Finjan differentiated Shaio as a "firewall patent" as opposed to "gateway" art based on a highly technical distinction that is contrary to this Court's order and Finjan's previous representation Finjan's invalidity expert stated that "[t]here is sort of a split between two different technologies that people put at their borders at their cite " (*Id.* at 1279:20-21) The first technology he described was a "firewall" that analyzed packets as they go from the server through the firewall to the client (*Id.* at 1279:18-1280:19.) The second technology he described was a "gateway." Finjan's expert stated:

> in a gateway, all the packets come to the gateway, so it can basically reassemble the program or reassemble the image or reassemble whatever it is you are transferring, do the analysis on it, and then allow that particular object through, if it thinks it is okay, or it drops the whole object if it thinks it is suspicious.

(*Id.* at 1281:7-13.) This definition of "gateway" was a blatant attempt to narrow the definition Finjan gave, and the Court accepted, at the *Markman* hearing.

Finjan's contradiction is clear. During the *Markman* stage, Finjan asserted that "gateway" has a very broad plain and ordinary meaning In particular, Finjan stated that a gateway, "much like a gate that stands between a house and the outside world" is a server that "stands between the client and the Internet[, and] most networks include a server that acts as a gateway in some fashion." Finjan Open. Markman Br. at 13 (D.I. 112).[9] Even Finjan's infringement expert used the broad definition of a gateway as an "intermediary between the client and the server." (Tr. at 336:9-10 (D.I. 228).) Finjan's invalidity expert, on the other hand, relied on this narrower definition of "gateway," in which it accepts and reassembles packets, in order to save the '194 patent from the prior art identified by Secure.

Finjan used this contradictory definition of "gateway" as the key to its validity defense Finjan's attorney stated that "there is a lot of *firewall patents* we are going to talk about." (Tr at 1287:24-25 (D I. 232) (emphasis added)) This improper distinction was used throughout Finjan's testimony defending the validity of its patents (*Id.* at 1291:2-4 (referring to Shaio as a "firewall patent"; *id.* at 1291:12-1292:2 (arguing that Shaio is a filtering firewall not a gateway); *id* at

[9] Further still, Finjan's counsel argued at the Markman hearing for an even broader application:

> Just to give you a very lay example, it is a very unambiguous term. We use the security downstairs. They are a gateway to this courtroom. You go through the security, and if you are carrying something they don't want you to carry, a gun or whatever, you are not getting through. The gateway is just that It is something that people go through to get somewhere else. And everyone knows what that is. And in a lay sense it could be something that simple

(Markman Tr. 13:10-18 (D I. 132).)

1294:25 (counsel referring to Shaio as a firewall patent); *id.* at 1314:4-12 (referring to Hershey prior art as a "firewall patent"). In fact, Finjan's expert expressly stated that because Shaio did not disclose a "gateway," based on his definition, Shaio did not anticipate.

> Q. So based on the sites we showed you in Shaio and the bytecode verifier that was incorporated by reference, do you have an opinion that any of the claims of the '194 patent are anticipated by Shaio?
> A. No. Once again, you know, there is a number of pieces that just sort of wipe it out. One, the *filtering firewall as opposed to a gateway*, which is one of these things we talked about, are two different animals.

(Tr. at 1317:3-10 (D.I. 232) (emphasis added).) Finjan's validity expert even relied upon the distinction between a filtering firewall and a gateway as the basis for his argument that the Finjan Patents met a long-felt need in the marketplace. (*Id.* at 1345:17-1347:9.) Finally, counsel made this "firewall patent" distinction the cornerstone of Finjan's validity argument in closing.

> The best prior art they could find to try to assert against us, the one they say anticipates, is the firewall patent. It has nothing to do with our technology. It is apples-and-orange time. I am telling you, it is completely difference. That is the Shaio reference, the Hershey reference, the Firewall Toolkit. All firewall patents.

(Tr. at 1614:10-16 (D.I. 233).)

Finjan must be bound by the broad interpretation of "gateway" that it spearheaded. Finjan cannot be free to argue for a broad construction at *Markman* and then narrow its construction in order to avoid prior art. Either the Court erred in rejecting Secure's request to define the term "gateway," or Finjan's attempt to narrow the Court's construction should be rejected as a matter of law. Regardless, this Court should grant Secure JMOL on the invalidity of the asserted claims of the '194 Patent, or at the very least, a new trial on the issue of invalidity of the '194 Patent.

Mr. Heberlein also argued that Shaio did not teach the limitation of "receiving an incoming Downloadable," because Shaio analyzes code within individual packets. But Dr. Wallach provided unrebutted testimony that an entire application can fall within one packet. (Tr. at 824:18-20 (D.I.

230).) And the Shaio Patent expressly teaches analyzing code within a packet "wherein said executable packet includes an applet." Shaio, col.6 ll.25-26 (Ex. 6). Thus, the evidence could lead a reasonable jury to only one conclusion: Shaio receives incoming Downloadables even under Mr. Heberlein's construction.

Because Finjan's arguments are based on a faulty claim construction, and it failed to rebut the evidence that Shaio analyzed certain Downloadables in their entirety, the Court should rule that Shaio anticipates the independent claims as a matter of law, or alternatively grant Secure a new trial on invalidity.

b. The Ji 1995 Patent (U.S. Pat. No. 5,623,600) In Combination with Chen (5,951,698) Or Lo 1994 Renders Obvious the Asserted Claims of the '194 Patent

Even if the Court was to accept Finjan's contradictory construction that a "gateway," as claimed in the '194 patent, is different then a "firewall," the Court should still grant JMOL of invalidity based on a combination of well-known antivirus techniques and a "gateway." As the Federal Circuit has recently made clear:

> When there is a design need or market pressure to solve a problem and there are a finite number of identified, predictable solutions, a person of ordinary skill has good reason to pursue the known options within his or her technical grasp. If this leads to the anticipated success, it is likely the product not of innovation but of ordinary skill and common sense.

KSR Int'l Co. v. Teleflex Inc., 127 S. Ct. 1727, 1742 (2007); *see also Agrizap, Inc. v. Woodstream Corp*, 2008 U.S. App. LEXIS 6471, at *13 (Fed. Cir. Mar. 28, 2008)("[t]he combination of familiar elements according to known methods is likely to be obvious when it does no more than yield predictable results.") Moreover, the Federal Circuit recently held that it is error to deny JMOL when there is a "strong prima facie case of obviousness." *Agrizap*, 2008 U.S. App. LEXIS 6471, at *17.

Unlike the dispute with Shaio over the scope of the term "gateway," the parties were able to agree that the Ji 1995 patent discloses performing known antivirus techniques on a "gateway" (Tr. at 1371:12-14, 19-25 (D.I. 232).) Finjan also did not, indeed it could not, rebut Dr. Wallach's testimony that analyzing executable code for suspicious computer operations in order to identify unknown viruses was a well-known antivirus technique prior to Finjan's application. (Tr. at 821:23-822:8; 862:8-862:15; 865:5-13; 948:10-948:21; 958:2-958:16 (D.I. 230, 231).)[10] For example, there was no dispute that the Lo 1994 article describes analyzing executable code for "tell-tale signs" of malicious behavior, by identifying computer operations that indicate a "File Read or File Write" may occur (Tr. 837:22-841:19 (D.I. 230).)[11] Likewise, there was no dispute that the Chen patent describes analyzing Macros for unknown viruses by identifying "suspect instructions" that are within the executable code (Tr. at 1379:21-1381:3 (D.I. 232).)

Finjan did not attempt to rebut that it was within the "technical grasp" and would not have involved "any particular challenge or difficulty" for one of ordinary skill to place these known antivirus techniques for stopping unknown viruses on a gateway as instructed by Ji 1995 (Tr. at 856:18-857:8; 957:17-23 (D.I. 230, 231).) Indeed, one of Finjan's technical experts stated that a gateway is "definitely the obvious place where you would put a protection system." (Tr. at 336:9-11 (D.I. 228).) Consequently, the Court should grant Secure JMOL of invalidity of the asserted claims of the '194 patent, because the jury had no reasonable basis to conclude that it would not have been obvious to combine these techniques, or a new trial

[10] *See, e.g.*, Franz Veldman, *Combating Viruses Heuristically*, Virus Bulletin Conference, 67-76 (Sept. 2003) (Ex. 10); Chen, U.S. Patent No. 5,951,698 (Ex. 8); Raymond W. Lo et al, *MCF A Malicious Code Filter* (May 4, 1994) (Ex. 11); Raymond Lo et al., *Towards a Testbed for Malicious Code Detection*, IEEE (1991) (Ex. 23)

[11] Raymond W. Lo et al, *MCF A Malicious Code Filter* (May 4, 1994) (Ex. 11).

i. Independent claims 1, 32, and 65

Mr. Heberlein's only bases for claiming that Ji 1995 and Chen do not render the independent claims of the '194 Patent obvious were (1) it does not extract information from a Downloadable to create a profile and (2) it does not use a security policy. But on cross examination, Mr Heberlein admitted that Chen discloses these limitations. First, Mr Heberlein admitted that Chen discloses extracting suspicious instructions from the Downloadable: "Chen extracts these -- looks for these suspicious computer operations in a downloadable." (Tr. at 1380:4-8 (D.I. 232) (emphasis added).) Second, Mr Heberlein admitted that Chen discloses using a security policy in response to the identified suspect instructions

> Q. So Chen says, If I find a virus, I am either going to delete that macro or fix it. Right?
> A. Yes, I don't disagree with that.
> Q. And deleting the macro or fixing it is a way of not permitting that macro to execute if it violates a *security policy*, isn't it?
> A. I would agree with that.

(*Id.* at 1380:15-1381:3 (emphasis added).)

Given the undisputed portions of Dr. Wallach's testimony and Mr. Heberlein's own admissions, the jury could only reasonably conclude that the combination of Ji 1995 and Chen or Lo 1994 render the asserted independent claims obvious. Consequently, this Court should grant JMOL to Secure regarding the invalidity of independent claims 1, 32, and 65 for reasons of obviousness based on the combination of Ji 1995 and Chen, or a new trial

ii Dependent Claims

The combinations of Ji 1995 with other prior art references also render the asserted dependent claims obvious as a matter of law.

With respect to claim 2 and 3, Mr. Heberlein made the same arguments as he did for the independent claims, so again, these same arguments apply for granting JMOL on these claims as exist for the independent claims.

With respect to claim 4, Dr. Wallach combines the Microsoft Authenticode reference and the Mueller patent to invalidate the claims. The Authenticode reference states that "firewall operators can also implement filters to accept signed software components from certain companies." (Ex. 12). Dr. Wallach's testimony that it would take a trivial amount of time for one of ordinary skill to combine this art with Authenticode also went unrebutted. (Tr. at 964:11-13 (D.I. 231).) This conclusion is unreasonable as a matter of law in light of the evidence. Consequently, the claim is invalid as obvious in light of this combination.

With respect to claims 5-7, these are all limitations based on URL filters. Dr. Wallach identified that the source code for the Firewall Toolkit (FWTK) includes a URL filter that allows one to specify particular URLs to allow or deny. (*Id.* at 964:23-965:17.) He also identified that this type of URL filter was well known in 1996 and easy to implement on a firewall. (Tr. at 964:23-965:17 (D.I. 231).) The testimony even showed that the very URL filter that Finjan accused of infringing this claim, SmartFilter, was announced prior to Finjan's patent application. (Tr. at 755:20-756:2 (D.I. 230).) Mr. Heberlein did not rebut Dr. Wallach's statement that these URL filters were well known and easy to implement on a firewall. (Tr. at 1359:10-1361:18 (D.I. 232).) Consequently, a reasonable jury could only conclude that these claims are rendered obvious.

Claims 8-11 and 33-36 include limitations based on specific types of Downloadable. All of these particular types of Downloadables are in the FWTK and Mr. Heberlein could not rebut the testimony. He simply disagreed by stating that he did not know what portion of the FWTK to which Dr. Wallach was referring. Dr. Wallach very specifically stated that HTTP gateway section

(Tr. at 858:22-859:1 (D.I. 230).) Mr. Heberlein's conclusory remark fails to amount to a legally sufficient basis to rebut Secure's invalidity contentions on these claims.

Mr. Heberlein's only rebuttal of Dr. Wallach opinion that Ji 1995 and Chen render claim 12 obvious was that the combination did not disclose extracting instructions for a profile But Mr Heberlein admitted that Chen discloses extracting instructions (Tr at 1380:4-8)

Dr. Wallach's validity contentions with respect to Claims 13 and 14 were also ignored by Mr. Heberlein. Mr. Heberlein simply concluded that he didn't know what Dr. Wallach's basis was for asserting invalidity. However, Dr. Wallach again pointed to the FWTK, which includes the user-based security policy. (Tr. at 966:15-967:2 (D.I. 231).)

Claims 24 and 26 are based on traditional anti-virus scanning techniques. Chen discloses this traditional technique. Chen, Col.2 ll.53-54 (Ex. 8). Likewise, claim 25 describes a whitelist to allow certain downloadables. As Dr. Wallach indicated, the FWTK contains a whitelist.

Claim 27 is specifically disclosed by the Hershey reference In fact, Hershey discloses a computer system that "provides methods and apparatus for immunizing a computer system ... against a subsequent infection by a previously unknown and undesirable software entity." Hershey, Col.5 ll.24-27 (Ex. 13). Mr. Heberlein did not disagree that Hershey disclosed this particular limitation Consequently, this limitation is invalid as a matter of law. (Tr. at 1331:23-1332:5 (D.I. 232).)

Claims 28-29 are again based on whitelisting or blacklisting URLs based on an administrative policy. These features of a URL filter were within the FWTK and Mr Heberlein failed to rebut the evidence that URL filters were well known in 1996

Finally, claim 30 requires notifying a user when there is a violation of the security policy Ji 1995 and Chen both disclose notifying a user. Ji 1995, Col.8 ll.28-31 (Ex. 7); Chen, Col.17 ll.21-23

(Ex. 8). Mr. Heberlein did not deny this, but instead relied on the invalidity of claim 1.

Based on the evidence, testimony, and admissions regarding the asserted dependent claims of the '194 Patent, no reasonable jury could have concluded that the claims were valid Consequently, the Court should grant Secure JMOL on invalidity of the asserted dependent claims, or grant a new trial.

B. <u>The Court Should Grant JMOL, Or In The Alternative A New Trial, On Invalidity Of The '780 Patent, If the Claim Allows Hashing a Downloadable and Its Components Separately</u>

If the Court ultimately determines that the '780 patent merely requires hashing a Downloadable separate from its referenced components, then Microsoft Authenticode and the Mueller patent, Ex. 14, each anticipate and/or render the asserted claims obvious The unrebutted testimony indicates that Authenticode and Mueller both disclose hashing Downloadables and software components. (Tr. at 977:14-980:1 (D.I. 231).) Consequently, if the Court reverses its previous claim construction and agrees that hashing together means together in time, then the Court should grant Secure JMOL of invalidity on the '780 patent, or a new trial

C. <u>The Court Should Grant JMOL, Or In The Alternative A New Trial, On Invalidity Of The '822 Patent</u>

Finjan's invalidity expert, Mr. Heberlein opined that Ji 1997, U.S. Pat. No. 5,983,348 (Ex. 15), does not perform the limitation of "determining whether the downloadable-information includes executable code." Ji 1997 clearly discloses identifying at least one type of executable code, Java Applets *See id.* at col 6 ll 53-55 ; col 3 ll-16-25. In fact, in the field of invention the inventor discloses that "[t]his invention pertains to computer networks and specifically to *detecting* and preventing operation of computer viruses and other types of *malicious computer code.*" *Id.* at col.1 ll 5-7 (emphasis added). This was the only limitation that Mr. Heberlein affirmatively argued is absent from the Ji 1997 patent (Tr. at 1341:2-1341:5 (D I. 232)). Consequently, the only reasonable conclusion the jury could reach was that the Ji 1997 patent anticipates the claims.

Moreover, claims 12 and 13 of the '822 patent contain several limitations that contain means-plus-function limitations without the requisite corresponding structure under 35 U.S.C § 112, ¶ 6 As discussed in Secure's *Markman* briefs, D I. 111, 120, several limitations are written in means-plus-function form because "even though the catch phrase is not used, [if] the limitation's language does not provide any structure, and [t]he limitation is drafted as a function to be performed rather than definite structure or materials" *Mas-Hamilton Group v LaGard, Inc*, 156 F.3d 1206, 1213 (Fed. Cir. 1998). These limitations include: content inspection engine, packaging engine, linking engine, transfer engine, MPC generator. Dr Wallach's testimony that these are not standard terms associated with standard structures was unrebutted. (Tr at 1001:11-1002:1; 1002:19-1003:13; 1003:21-1004:6 (D I. 231)). This lack of corresponding structure means the Court should grant JMOL of invalidity on claims 12-13 of the '822 patent do not satisfy 35 U.S.C. 112, ¶ 6. *See Finisar Corp v DirecTV Group, Inc.*, 2008 U S App LEXIS 8404, at *43-44 (Fed. Cir. Apr. 18, 2008).

VI. THE COURT SHOULD GRANT JMOL, OR IN THE ALTERNATIVE A NEW TRIAL OR REMITTITUR, REGARDING DAMAGES RELATED TO FINJAN'S PATENTS

A. The Jury's $9.18 Million Verdict Is Unreasonable, Excessive and Shocks The Conscience, [redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

B. The Royalty Rates Of 16% For Software and 8% For Hardware Appliances Are Unreasonable, Erroneous And Excessive

1. Finjan's Damages Expert's Erroneous And Deceptive Application of the "Rule of Thumb" Led To Unreasonable And Excessive Royalty Rates

Mr. Parr misapplied and manipulated the application of the "rule of thumb" analysis, such that his opinion provided the jury with unreasonable, erroneous and excessive royalty rates At the outset, Mr. Parr and Secure's damages expert, Carl Degen, agreed that: (1) the "rule of thumb" should be applied in determining the royalty rate; (2) the "rule of thumb" takes a quarter to a third of expected net operating profits as a starting point in determining a royalty rate; and (3) product-specific operating profit numbers are preferred for the analysis (Tr. at 654:4-656:7; 656:17-19; 1130:17-24; 1123:2-25; 1113:14-24 (D.I. 229, 231).) However, Mr. Parr failed to properly apply the methodology he purported to use, and mislead the jury into finding a punitive royalty amount that is in fact 100% of Secure's operating profit[14] Finjan should not be allowed to profit from the

[14] Contrary to law and this Court's jury instructions, Finjan based its entire royalty rate case on a theory of punitive damages. In Finjan's opening statement, Finjan told the jury it was

fruits of Mr. Parr's misapplication and manipulation – JMOL, or a new trial or remittitur, is necessary.

First, Mr. Parr ignored Secure's product-specific operating profit financials, which were kept in the ordinary course of business, and instead used the company-wide gross profits reported in public financial documents. (Tr. at 655:4-668:22(D.I. 229).) Mr. Parr even admitted he had possession of Secure's business records showing product-specific net operating profit. (*Id.*). These were the same business records that Mr. Degen used to properly calculate a 16% product-specific net operating profit and a 4% royalty rate. (Tr. at 1117:23-1138:19 (D.I. 231); Exs. 19-21 (DTX 1319; DTX 1320; DTX 1321).). Finjan never rebutted or impeached these product-specific financial documents or Mr. Degen's 16% operating profit opinion.

Then, Mr. Parr admitted, after ignoring the real product-specific financials, he manipulated and "adjusted" the company-wide financials to attempt to determine operating profit, which raised the royalty rate. (Tr. at 656:17-657:15 (D.I. 229); *see also* 661:9-662:25.)

Even using Mr. Parr's inflated profit margin numbers from the company-wide financials, the royalty rate should be only 3-5%. However, Mr. Parr inflated even that rate by erroneously considering financials for 2004 and 2005, years in which Secure did not even sell the accused product. (*Id.* at 657:21-659:15 (D.I. 229).) And then, adding insult to injury, Mr. Parr considered a wholly inappropriate alleged 99% <u>gross profit</u> number (as opposed to net operating profit), <u>not related to the Webwasher product</u> (as opposed to product-specific), and pulled from an excerpt from a deposition (as opposed to business record financials), to boost his high royalty rate even more. (*Id.* at 626:2-20; 667:1-13; 680:8-683:18 (D.I. 229); 1132:9-1134:5 (D.I. 231).)

asking for a high royalty rate because Secure allegedly copied Finjan. (Tr. at 126:19-127:16 (D.I. 227).) A reasonable royalty should not be punitive in nature. Apparently, in awarding its excessively high verdict, the jury followed this legally flawed theory and Mr. Parr's equally flawed "rule of thumb" analysis. Justice requires that this type of verdict be overturned.

Based on these "adjustments" and misapplications of the "rule of thumb," Mr Parr found grossly overstated operating profits of 55% for Webwasher software 25% for Webwasher hardware. (*Id* at 625:4-6; 626:3-20.) These rates are clearly contradicted by the actual, non-adjusted, product-specific operating profit financials. In fact, when applied to those actual product-specific financials, Mr. Parr's rate constitutes 100% of Secure's actual product-specific operating profits Such a grossly excessive and unreasonable verdict cannot stand Instead, if Finjan is to be awarded any damages, the substantial evidence supports Mr Degen's correct "rule of thumb" analysis, which leads to reasonable royalty rates of 4% for Webwasher Software and Hardware Appliances, and 1% for Cyberguard TSP Hardware Appliances (Tr at 1130:20-1138:19 (D I 231).) The Court should grant JMOL by applying Mr. Degen's royalty rates, or grant a new trial or remittitur to remedy the excessively high royalty rates.

2. No Reasonable Jury Would Find An 8% Royalty Rate For Cyberguard TSP Hardware Appliances

An 8% royalty rate for Cyberguard TSP Hardware Appliances is not supported by sufficient or substantial evidence, is excessive on its face, and shocks the conscience. As explained in Section IV E., the substantial evidence is that the accused Webwasher functionalities are locked, not enabled and not available on Cyberguard TSP. (Tr. at 712:2-7; 713:13-21; 714:1-6 (D I 230); 1115:7-1116:4 (D I. 231).) Therefore, there is very limited value, if any, to having Webwasher code on Cyberguard TSP.

According to Mr. Degen, the only value of Webwasher on Cyberguard TSP was Secure's ability to "preserve some customer relationships and switch them over to the Sidewinder product." Thus, Secure "would have been willing to pay something" but "certainly a four-percent royalty for that use of it would be too high." (Tr. at 1115:7-16 (D I. 231).) Mr Degen opined that a 1% percent royalty would be appropriate for Cyberguard TSP (*Id* at 17-19.)

In the context of the hypothetical negotiation, no reasonable jury could find that a willing licensee would agree to an 8% royalty for a product that has never been sold and will never be sold with the accused function turned on. In comparison, the jury awarded an 8% royalty for Webwasher Hardware Appliances – on which there is no dispute that the Webwasher software is available (on a module by module basis) for purchase. It shocks the conscience for the jury to award the same royalty rate for Cyberguard TSP. Secure requests JMOL that a 1% royalty be applied to Cyberguard TSP sales, or in the alternative a new trial or remittitur.

C. **The Royalty Base Is Not Supported By Sufficient Evidence, And Is Erroneous And Excessive**

1. Foreign Sales Should Be Excluded

The jury erroneously disregarded the law regarding foreign sales and the substantial evidence regarding Secure's foreign sales, and included foreign sales in their damages calculation.

The jury was properly instructed on the law regarding foreign sales. *See* D.I. 225, p. 15. Finjan has not challenged this instruction in any post trial motion. Mike Gallagher, Secure's Sr. VP of Product Development and Customer Support, provided substantial testimony regarding the foreign sales operations of Cyberguard and Webwasher. (Tr. at 708:6-709:24 (D.I. 230).) Secure's damages expert, Mr. Degen, provided substantial and unrebutted testimony regarding the amount of foreign sales of the accused products that should be deducted from the royalty base. (Tr. at 1153:3-25; 1157:1-11 (D.I. 231); Exs. 4-5 (Degen Figures 2 and 2A).) Mr. Parr did not take into account foreign sales of accused products and did not rebut Mr. Degen's foreign sales opinions. (*Id.* at 669:15-19; 1154:1-4 (D.I. 229, 231).)

Despite the proper jury instruction and Secure's substantial evidence, the jury included foreign sales, as is evidenced by the fact that the jury awarded the royalty base provided by Finjan's expert Mr. Parr. For instance, Mr. Parr opined that the Webwasher software royalty base is $49 million (Tr at 645:6-8 (D.I. 229), and the jury awarded $49 million for the Webwasher software royalty base. The jury clearly disregarded the jury instructions and un-rebutted testimony of Mr. Gallagher and Mr. Degen. This was error and unreasonable Therefore, the Court should grant JMOL that no foreign sales should be included in the damages award, or in the alternative a new trial on foreign sales or remittitur subtracting foreign software sales in the amount of $13,594,697 from the damages award. (Tr. at 1153:3-25; 1157:1-11 (D I. 231); Exs. 4-5 (Degen Figures 2 and 2A).)

2. Sales To The U.S. Government Should Be Excluded

The jury erroneously disregarded the law regarding government sales and the substantial and unrebutted evidence regarding Secure's government sales, and included government sales in their damages calculation. The jury was properly instructed on the law regarding sales to the U.S Government. *See* D.I. 225, p. 42. Mr. Gallagher provided substantial and unrebutted testimony regarding Secure's government sales. (Tr. at 709:25-710:25 (D.I 230).) Mr. Degen provided substantial and unrebutted testimony regarding the amount of government sales of accused products that should be deducted from Mr. Parr's base. (Tr. at 1154:22-1155:2 (D I 231); Exs 4-5 (Degen Figures 2 and 2A).) The jury disregarded this instruction and included sales to the U.S. Government in their damages calculation. For instance, again, the jury accepted Mr. Parr's $49 million royalty base for Webwasher software, without properly subtracting government sales Therefore, the Court should grant JMOL that no government sales should be included in the damages award, or in the alternative a new trial on government sales or remittitur subtracting $1,150,129 in government sales[15] from the damages award.

[15] The amount of government sales may be larger, since this number reflects the sales after Mr Degen's other sales deductions.

3. Sales Of Accused Products On Which Alleged Infringing Technology Is Locked, Not Enabled, And/Or Otherwise Not Available Should Be Excluded

As explained above (Section IV E), it was error for the Court to reject Secure's Proposed Jury Instruction No. 19 2. For the same reasons stated above, a reasonable and properly instructed jury would have found that sales accused products with the accused functionalities locked, not enabled, and/or unavailable, should not be included in the damages award.[16] The Court should grant JMOL, or in the alternative a new trial or remittitur

4. Finjan Failed To Present Sufficient Evidence Regarding The Cyberguard TSP Hardware Appliances Royalty Base

Even if Cyberguard TSP with Webwasher is held to infringe, the jury's finding of a $13 5 million royalty base for Cyberguard TSP Hardware Appliances should be set aside, or at the very least re-tried or reduced, because Finjan presented insufficient evidence to show that there was $13.5 million in sales of Cyberguard TSP with Webwasher. Finjan's damages expert Mr. Parr opined that there were $12 million, not $13 5 million, in sales of Cyberguard TSP (Tr at 646:16-25 (D.I. 229).) Thus, even taking Mr. Parr's Cyberguard TSP base, the jury's verdict was erroneous and excessive. There is no evidence in the record to support an additional $1.5 million added on to Mr. Parr's royalty base for Cyberguard TSP, and the Cyberguard TSP royalty base should be deducted by at least $1.5 million.

But the verdict should be overturned, or reduced by more than $1 5 million, because: (1) Mr. Parr testified that only sales of Cyberguard TSP products that have the "Webwasher component" should be included in the Cyberguard TSP royalty base (*Id.* at 634:1-11) ; (2) Finjan

[16] Even if the Court finds that these sales are "offers for sale," as Finjan may suggest, Finjan never offered any opinion or testimony regarding the value of an "offer for sale" of an accused product with the locked accused functionalities. Therefore, Finjan provided insufficient evidence to support an award of damages for "offers for sale" of the accused functionalities.

presented no evidence that anything other than Cyberguard TSP version 6 4 has ever included "Webwasher component" (*Id.* at 635:14-24); and (2) Finjan presented no evidence linking version 6.4 to $12 million in sales At best, Mr. Parr's Cyberguard TSP royalty base is conclusory and unsupported, if not excessive. The Court should grant JMOL, or in the alternative a new trial or remittitur, on the Cyberguard TSP royalty base

D. If This Court Grants JMOL Or A New Trial On Invalidity Or Nonifringement, Then the Court Should Order A New Trial On Damages

In this case, the jury did not reveal the means by which it calculated damages, nor did it indicate how much of the damages it apportioned to a particular patent or claim. Consequently, in the event the Court grants JMOL or a new trial on noninfringement or invalidity, the Court should grant a new trial on damages *See Memphis Cmty. Sch Dist v. Stachura*, 477 U.S 299, 312 (1986) (granting a new trial because an instruction was erroneous and the jury did "not reveal the means by which the jury calculated damages"); *see also Verizon Servs Corp v Vonage Holdings Corp*, 503 F 3d 1295, 1310 (Fed. Cir 2007) (vacating a damages award in a multi-patent case when a new trial was granted on infringement of one of the patents).

VII. THE COURT SHOULD GRANT JMOL, OR IN THE ALTERNATIVE A NEW TRIAL, ON NO WILLFUL INFRINGEMENT

A. Legal Standards To Prove Willful Infringement

No reasonable jury could find that Secure willfully infringed the '194, '780, and '822 Patents, especially in light of the heightened standard for willful infringement set forth by *In re Seagate Tech., L.L.C.*, 497 F.3d 1360 (Fed Cir. 2007). In overruling the lower threshold that existed under previous case law, the Federal Circuit held that proof of willful infringement requires "at least a showing of objective recklessness." *Id* at 1370-71; *see also* Final Jury Instruction No 19 (D.I. 225) Under *Seagate*, the patentee must meet a two-step test, by clear and convincing evidence: (1) a patentee must show by clear and convincing evidence that the alleged infringer acted despite an objectively high likelihood that its actions constituted infringement of a valid patent. The state of mind of the accused infringer is not relevant to this objective inquiry; and (2) if the threshold objective standard in (1) is satisfied, the patentee must also demonstrate that this objectively-defined risk (determined by the record developed in the infringement proceeding) was either known or so obvious that it should have been known to the accused infringer. *Id.* at 1371. An alleged infringer does not have an "affirmative duty of care" or an "affirmative obligation to obtain opinion of counsel" to ensure that he does not infringe a patent. *Id.* Since *In re Seagate* was decided, at least four courts have reversed a jury's verdict of willful infringement. *See Finisar Corp.*, 2008 U.S. App. LEXIS 8404, at *38-40 (reversing jury's willfulness verdict); *Innogenetics, N.V. v. Abbott Labs.*, 512 F.3d 1363, 1381 (Fed. Cir. 2008)(affirming district court's grant of JMOL reversing jury's willfulness verdict); *Trading Techs. Int'l, Inc. v. eSpeed, Inc.*, 2008 U.S. Dist. LEXIS 295, at *10 (N.D. Ill. Jan. 3, 2008) (granting defendant JMOL to reverse jury's willfulness verdict); *TPIG, Inc. v. AT&T Corp.*, 2007 U.S. Dist. LEXIS 79919, at *35-37 (E.D. Tex. Oct. 29, 2007)(granting defendant JMOL to reverse jury's willfulness verdict).

B. <u>Finjan Failed To Present Sufficient Evidence To Prove By Clear And Convincing Evidence That Secure Willfully Infringed Finjan's Patents</u>

The jury's finding that Secure willfully infringed the '194 Patent, '780 Patent, and '822 Patent, is not supported by sufficient or substantial evidence adequate to support Finjan's burden of proof of clear and convincing evidence that Secure acted with reckless disregard of Finjan's Patents. No reasonable jury could find that Finjan proved by clear and convincing evidence that Secure willfully infringed, and such a finding is against the weight of the evidence

1. Finjan Failed Present Clear And Convincing Evidence to Meet *Seagate's* Objective Factor Because Secure Presented Substantial Defenses of Non-Infringement and Invalidity

As mentioned above, under *Seagate's* objective prong, the "patentee must show by clear and convincing evidence that the infringer acted despite an objectively high likelihood that its actions constituted infringement of a valid patent." 497 F 3d at 1371 The assertion of substantial litigation defenses has, even before *Seagate*, been a basis for finding non-willfulness. *Ajinomota Co v. Archer-Daniels-Midland Co*, 228 F.3d 1338, 1351-52 (Fed. Cir. 2000); *Electro Med. Sys., S A v Cooper Life Scis, Inc.*, 34 F.3d 1048, 1057 (Fed Cir. 1994). The significance of litigation defenses in willfulness determinations is only enhanced by the heightened *Seagate* standard *See Black & Decker, Inc v Robert Bosch Tool Corp*, 2008 U.S. App. LEXIS 207, at *18 (Fed Cir Jan. 7, 2008)(stating that under *Seagate* "both legitimate defenses to infringement claims and credible invalidity arguments demonstrate the lack of an objectively high likelihood that a party took actions constituting infringement of a valid patent"). Further, even where a patent if found to be valid and infringed, it is appropriate to find no willful infringement where such defenses are "substantial, reasonable, and far from being the sort of easily-dismissed claims that an objectively reckless infringer would be forced to rely upon." *ResQNet com v Lansa, Inc.*, 533 F. Supp 2d 397, 420 (S.D.N.Y. 2008).; *see also Franklin Elec Co. v Dover Corp.*, 2007 U S. Dist LEXIS 84588, at *23 (W.D Wis. Nov. 15, 2007)(denying defendant's motion for summary judgment on non-infringement, but finding no willful infringement, "[g]iven the significant support in the language of the patent, the specification, and the prosecution history for defendant's non-infringement position.")

As explained above, Secure presented substantial evidence at trial to prove that Finjan's Patents are not infringed and are invalid. (*See* Sections IV and V.) Based on the arguments presented here, Secure has both legitimate defenses to Finjan's infringement claims and credible invalidity arguments. Therefore, no reasonable jury could find in favor of Finjan on *Seagate's* objective prong. *See Black & Decker*, 2008 U.S. App. LEXIS 207, at *18; *see also Trading Techs.*, 2008 U.S. Dist. LEXIS 295, at *7-8 (finding no willful infringement because defendant "sufficiently asserted defenses to infringement and those defenses were neither unreasonable nor frivolous.").

Even if the Court denies Secure's post-trial motion regarding infringement and invalidity, the Court should still overturn the willfulness verdict because Secure's non-infringement and invalidity defenses are still substantial, reasonable, and far from being the sort of easily-dismissed claims that an objectively reckless infringer would be forced to rely upon. *See ResQNet.com*, 533 F. Supp. 2d at 420. Infringement and validity of Finjan's Patents have been hotly contested in this case. *See Trading Techs.*, 2008 U.S. Dist. LEXIS 295, at *7-8 (finding no willful infringement where the "validity of plaintiff's patents has been hotly contested.") Indeed, as the Court acknowledged, the parties engaged in a "heated trial" over infringement and validity of Finjan's Patents. (Tr. at 1211:18 (D.I. 232).) Secure presented eleven strong prior art references. (D.I. 225, pp. 24-25.) Secure's non-infringement and invalidity expert took the stand for well over one full trial day. (*See* Tr. at 775:22-1103:22 (D.I. 230-231).) Perhaps most telling of how strong Secure's non-infringement and invalidity defenses - Finjan hired three separate experts, Dr. Bishop, Dr. Vigna, and Mr. Heberlein, to defend against Secure's non-infringement and invalidity defenses. (Tr. at 174:21-201:3; 322:17-573:7; 1271:1-1451:2; (D.I. 227-29; 232-33).) Secure presented significant support in the language of the patents, the specifications, and the prosecution histories in support non-infringement positions. *See Franklin*, 2007 U.S. Dist. LEXIS 84588, at *23. And, the jury ultimately rejected Finjan's argument that Secure's products literally infringe claim 3 of the '194 Patent and all of the asserted claims of the '780 Patent (D.I. 226, 242.) Instead, the jury only found infringement under the doctrine of equivalents for those claims (*Id.*) And, Secure's invalidity defenses are strong, especially in light of the new standard for obviousness under *KSR Int'l Co v Teleflex Inc.*, 127 S. Ct. 1727, 1742 (U.S. 2007) In light of Secure's substantial, reasonable, credible, and legitimate non-infringement and invalidity defenses, it was unreasonable for the jury to conclude that Secure willfully infringed Finjan's Patents. The Court should grant JMOL, or a new trial

2. Finjan Failed To Present Clear And Convincing Evidence To Meet *Seagate's* Subjective Factor

a. Finjan Failed To Prove That Secure Acted With Knowledge Of Finjan's Patents

Because Finjan failed to present sufficient evidence to prove *Seagate's* objective prong, the jury should never have determined whether Finjan proved the second, subjective, prong. However, even if Finjan could present substantial evidence that there was a high likelihood of infringement of a valid patent, the following make it was unreasonable for a jury to conclude that Secure acted with knowledge of Finjan's Patents:

- Finjan has failed to put forth a single piece of evidence that anyone at Secure was on notice of the '780 and '822 Patents prior to the filing of this lawsuit.
- Finjan introduced only one Webwasher document that references the '194 Patent (PTX-38). That same document includes references to fourteen other patents. *Id.* These other patents even included patents that Secure is relying on to invalidate the Finjan patents in this case.
- Finjan put forward no trial testimony from Roland Cuny, the author of the only document referencing the '194 Patent. (PTX-38)
- The only trial testimony on PTX-38 was Martin Stecher, as follows:

  Q: Is this document a summary of the research that Roland Cuny was doing?

> A: Almost everything that Mr Cuny did was documented within the intranet by Mr. Cuny himself; and regard to this task, he saved or he put this content on intranet "
>
> (Tr. at 321:1-8 (D I. 228).)

- Finjan failed to present any evidence that anyone other than Mr. Cuny had access to or read the intranet described above, much less any evidence that anyone involved with the development of proactive scanning had read PTX-38.
- Finjan failed to make any connection between the author of PTX-38, Roland Cuny, and the development of proactive scanning
- Roland Cuny's inclusion of the '194 Patent on a long list of patents does not demonstrate that Mr Cuny had any reason to believe that Webwasher may infringe the '194 Patent There is no evidence that he performed an infringement analysis or spoke with anybody regarding potential infringement of the '194 Patent. At best, PTX-38 proves that Mr Cuny was aware of prior art that could invalidate Finjan's Patents.
- Finjan failed to present any evidence that any of the engineers involved in the development of the Webwasher product and proactive scanning had any awareness of the '194 Patent

Based on the enormous failure of proof described above, no reasonable jury could find that Secure acted with knowledge of Finjan's Patents, such that any objectively-defined risk of infringement was either known or so obvious that it should have been known to it.

b. Secure Did Not Copy

At trial, Finjan made much of its alleged copying case. However, alleged copying is only relevant once *Seagate's* threshold of "objective likelihood of infringement" is met *Trading Techs*, 2008 U.S. Dist LEXIS 295, at *8-9. As described above, no reasonable jury could find that Secure acted despite an objectively high likelihood that its actions constituted infringement of a valid patent, and further no reasonable jury could find Secure acted with knowledge of Finjan's Patents Therefore, no reasonable jury could find that Secure copied Finjan's Patents However, even if Finjan could prove that Secure met *Seagate's* objective prong and that Secure had knowledge of Finjan's Patents, still no reasonable jury could find copying

In response to Secure's Rule 50(a) motion JMOL at trial, Finjan erroneously argued that: (1) Mr. Stecher, Mr. Berzau, and Mr. Alme looked at Finjan's Patents and then developed the Webwasher product; and (2) Secure considered two options, one of which was to copy Finjan, and that Secure chose the option to copy (Tr. at 688:16-689:15 (D.I. 230).) The substantial evidence at trial, however, proved just the opposite — Mr Stecher, Mr Berzau, and Mr. Alme never looked at Finjan's Patents, and Secure did not choose the option to copy.

Finjan presented no evidence or testimony that Mr. Stecher looked at Finjan's Patents. And, with regard to the copying option, Mr. Stecher testified as follows:

> Q: Which of these options did Webwasher pursue?
>
> A: **<u>None of them was implemented</u>**.

(Tr. at 319:1-3 (D.I. 228)(emphasis added)) Instead, Mr Stecher testified, Secure chose "more diverse methods" and a "rules and category-based system with media types" that was implemented based on further meetings to improve the performance rate of the product. (*Id.* at 319:4-12.)

Finjan presented no evidence that either Mr. Alme or Mr. Berzau looked at Finjan's Patents Mr Alme testified that he did not review any other products while developing the proactive scanner of Webwasher (*Id.* at 305:23-25.) Mr. Alme testified that he merely saw screenshots of Finjan's product (*Id* at 306:12-16.) Mr. Berzau testified that he merely saw the Finjan "GUI" and it did not include proactive scanning. (*Id* at 580:2-4 (D.I. 229))("Was one of the configuration options ProActive scanning? Answer: No.") [17]

[17] Even if Mr. Alme or Mr. Berzau did copy Finjan's product (as opposed to Finjan's patents), it would not be relevant to willful infringement. To prove that copying a product is relevant to willful infringement, Finjan must prove that its product is an embodiment of the patents. Finjan did not make any attempt to prove that its product was an embodiment of the patent. Finjan did not have any expert do a limitation-by-limitation analysis of the product to prove that Finjan's products fell within the three patents in suit. Consequently, even if Secure had copied Finjan's product, it would be wholly irrelevant to the willful infringement analysis.

Finjan's validity expert Todd Heberlein admitted that, in determining whether Secure copied Finjan, he never looked at Webwasher source code, he never looked at the Webwasher product, he does not know how the Webwasher product operates, and he never did a limitation by limitation analysis of Secure's products compared to Finjan's Patents. (Tr. at 1352-55 (D.I. 232).) In fact, to the extent Finjan attempts to argue that Secure copied Finjan's product, none of Finjan's four technical experts compared the parties' source codes to determine if they matched.

Finally, the Webwasher Product Meeting Minutes, dated June 1, 2004, which discuss planning for Webwasher version 5.1, prove conclusively that Secure did not copy. (Ex. 22)(DTX-1056.) That document states unequivocally that the copying option was "dropped." (*Id. at* p. 1.)

Finjan failed to prove copying. Based on the substantial evidence and testimony discussed above, including but not limited to testimony from Mr. Stecher, Mr. Alme, Mr. Berzau, Mr. Heberlein, and DTX-1056, no reasonable jury could find that Secure copied. The Court should grant JMOL, or in the alternative a new trial.

VIII. THE COURT SHOULD GRANT JMOL, OR IN THE ALTERNATIVE A NEW TRIAL, ON INFRINGEMENT AND DAMAGES RELATED TO SECURE'S '361 PATENT

The Court should grant JMOL, or in the alternative a new trial, on Secure's claim that Finjan infringes the '361 Patent; and that Secure should be awarded damages in the amount of $88,585. (Tr. at 1166:4-1167:12 (D.I. 232).)

There is no factual dispute regarding how the Finjan NG products work. The only questions to resolve involve the legal matter of the correct claim scope of claim 15 of the '361 patent. Finjan applied, and the jury relied on, an improperly narrow interpretation of claim scope. Consequently, this Court should grant Secure JMOL that Finjan's NG products infringe Claim 15 of the '361 patent.

The first noninfringement argument involves reconciling the scope of the term "firewall" in the patent Finjan took the position that Finjan's NG appliances are not "firewalls" because a "firewall" does not control information at the application level (Tr at 1469:23-1470:23 (D I 233).) The patent, however, states that "[f]irewalls can control access at a network level, application level, or both." '822 patent, col.1, ll.30-31. Consequently, Finjan's argument should be rejected as a matter of law.

The second noninfringement argument involves whether claim 15, a product claim, requires the performance of any steps. Finjan's expert, Dr Jaeger, took the position that in order to infringe claim 15 there must be a step of querying a directory after receiving a network request (Tr. at 1471:14-1474:14 (D I. 233).) But as a matter of claim construction, claim 15 only requires the existence of first, second, and third portions of program code to infringe There is no legal requirement that the code is executed in some particular order. Consequently, the argument that the "first computer readable program code" must be executed before the "second computer readable medium program" is incorrect as a matter of law

The third noninfringement argument involves whether the claim limitation "authorization filter that is generated based on a directory schema" requires that the authorization filter actually contain the exact same schema as the directory. Claim 15 states that the authorization filter must be generated *based on* a directory schema Dr. Jaeger concedes that the NG appliances contains an authorization filter that is generated by importing information from a directory. (*Id.* at 1487:14-24). While the two parties do not dispute that the authorization filter in the NG appliance itself does not import the field names from a directory, it is undisputed that the authorization filter is generated by querying particular fields in a directory This is all that is required by the claims, because the authorization filter merely needs to be created *based on* the directory schema

The final noninfringement argument is based on a faulty assumption that the Claim 15 preamble is limiting. Finjan argues that the preamble of "[a] computer program product for enabling a processor in a computer system to implement an authentication process" means that the claim requires performance of authentication. But the preamble is not a limitation. It is black-letter law that "[a] preamble is not limiting, where a patentee defines a structurally complete invention in the claim body and uses the preamble only to state a purpose or intended use for the invention." *Symantec Corp. v. Computer Assocs. Int'l, Inc.*, 2008 U.S. App. LEXIS 7826, at *11 (Fed. Cir. Apr. 11, 2008). This is expressly an intended purpose and not a limitation.

Consequently, this Court should rule that Finjan's noninfringement arguments all depend on improper claim constructions and grant Secure JMOL of infringement of claim 15 of the '361 patent and grant Secure a new trial on damages of the '361 patent. Finjan did not dispute Secure's damages evidence and calculations with respect to the '361 Patent. If the Court grants Secure's JMOL motion regarding infringement of the '361 Patent, the Court should also order JMOL on damages to Secure in the amount of $88,585, or otherwise order a new trial.

CONCLUSION

Based on the foregoing, Secure respectfully requests that the Court grant Secure's motion under Rule 50(b), for JMOL, and under Rule 59, for a new trial or to alter or amend a judgment (remittitur).

Kelly E. Farnan

Fredrick L. Cottrell, III (#2555)
cottrell@rlf.com
Kelly E. Farnan (#4395)
farnan@rlf.com
Richards, Layton & Finger
One Rodney Square, P.O. Box 551
Wilmington, DE 19899
(302) 651-7700
Attorneys For Defendants-Counterclaimants

OF COUNSEL:
Ronald J. Schutz
Jake M. Holdreith
Christopher A. Seidl
Trevor J. Foster
Robins, Kaplan, Miller & Ciresi L.L.P.
2800 LaSalle Plaza
800 LaSalle Avenue
Minneapolis, MN 55402

Dated: April 25, 2008

UNITED STATES DISTRICT COURT
DISTRICT OF DELAWARE

CERTIFICATE OF SERVICE

I HEREBY CERTIFY that on April 25, 2008, I electronically filed the foregoing with the Clerk of Court using CM/ECF and caused the same to be served on the plaintiff at the addresses and in the manner indicated below:

E-MAIL AND HAND DELIVERY

Philip A. Rovner
Potter Anderson & Corroon LLP
1313 N. Market Street,
Hercules Plaza, 6th Floor
Wilmington, DE 19899-0951

I further certify that on April 25, 2008, the foregoing document was sent to the following non-registered participants in the manner indicated:

E-MAIL

Paul J. Andre
Lisa Kobialka
King & Spalding, LLP
1000 Bridge Parkway, Suite 100
Redwood Shores, CA 94065

Kelly E. Farnan
Kelly E. Farnan (#4395)

UNITED STATES DISTRICT COURT
DISTRICT OF DELAWARE

CERTIFICATE OF SERVICE

I HEREBY CERTIFY that on May 2, 2008, I electronically filed the foregoing with the Clerk of Court using CM/ECF and caused the same to be served on the plaintiff at the addresses and in the manner indicated below:

HAND DELIVERY

Philip A. Rovner
Potter Anderson & Corroon LLP
1313 N. Market Street,
Hercules Plaza, 6th Floor
Wilmington, DE 19899-0951

I further certify that on May 2, 2008, the foregoing document was sent to the following non-registered participants in the manner indicated:

FEDERAL EXPRESS

Paul J. Andre
Lisa Kobialka
King & Spalding, LLP
1000 Bridge Parkway, Suite 100
Redwood Shores, CA 94065

Kelly E Farnan
Kelly E. Farnan (#4395)

EXHIBIT 1

03/30/00

ISSUE CLASSIFICATION	
Class	Subclass
713	181

PATENT NUMBER 6804780

U.S. UTILITY Patent Application

5 ✓

O.I.P.E.	PATENT DATE
SCANNED JTV Q.A. MC	OCT 12 OCT 12 2004

APPLICATION NO.	CONT/PRIOR	CLASS	SUBCLASS	ART UNIT	EXAMINER
09/539667	1D	713	181	~~2785~~ 2131 ~~2151~~ ~~2184~~	Revak

APPLICANTS Shlomo Touboul

TITLE System and method for protecting a computer and a network from hostile downloadables

PTO-2040 12/99

ISSUING CLASSIFICATION

ORIGINAL		CROSS REFERENCE(S)						
CLASS	SUBCLASS	CLASS	SUBCLASS (ONE SUBCLASS PER BLOCK)					
713	181	713	201	176				
INTERNATIONAL CLASSIFICATION		717	178					
H04L	9/00							
G06F	11/30							

□ Continued on Issue Slip Inside File Jacket

9/10/04 Formal Drawings (10 shts) set ✓ 3/30/00

☒ TERMINAL DISCLAIMER	DRAWINGS			CLAIMS ALLOWED	
	Sheets Drwg.	Figs. Drwg.	Print Fig.	Total Claims	Print Claim for O.G.
	10	10	8	18	1

□ The term of this patent subsequent to ____________ (date) has been disclaimed.

☒ The term of this patent shall not extend beyond the expiration date of U.S Patent. No. 6,092,194

□ The terminal ____ months of this patent have been disclaimed.

Christopher Revak 5/13/04
(Assistant Examiner) (Date)

AYAZ SHEIKH
SUPERVISORY PATENT EXAMINER
TECHNOLOGY CENTER 2100
(Primary Examiner) 5/24/04 (Date)

Brenda Hannon 7/04
(Legal Instruments Examiner) (Date)

NOTICE OF ALLOWANCE MAILED
06-4-04

ISSUE FEE IM

Amount Due	Date Paid
$ 665.00	9/3/04

ISSUE BATCH NUMBER

WARNING:
The information disclosed herein may be restricted. Unauthorized disclosure may be prohibited by the United States Code Title 35, Sections 122, 181 and 368. Possession outside the U.S. Patent & Trademark Office is restricted to authorized employees and contractors only.

Form PTO-436A (Rev. 8/99)

FILED WITH: □ DISK (CRF) □ FICHE □ CD-ROM
(Attached in pocket on right inside flap)

(FACE)

Joint Trial Exhibit
JTX-52
Case No. 06-369 GMS

#6A/8-9-03

Attorney's Docket No.: 43426.00011 *PATENT*

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF MAILING

I hereby certify that this paper (along with any paper referred to as being attached or enclosed) is being deposited with the United States Postal Service on the date shown below with sufficient postage as first class mail in an envelope addressed to the Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450, on

Date: July 31, 2003 By: EM Janikowski
Eileen M. Janikowski

In Re Patent Application of:)
) Examiner: Christopher A. Revak
Shlomo Touboul)
) Art Unit: 2131
Application No: 09/539,667)
)
Filed: March 30, 2000)
)
For: SYSTEM AND METHOD FOR)
PROTECTING A COMPUTER)
AND A NETWORK FROM)
HOSTILE DOWNLOADABLES)
)

Assistant Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

AMENDMENT AND RESPONSE TO OFFICE ACTION UNDER 37 C.F.R. §1.111

Sir:

In response to the Office Action dated July 1, 2003 and pursuant to 37 C.F.R. §1.111, applicant respectfully requests that the above-identified application be amended as follows:

A

IN THE ABSTRACT OF THE DISCLOSURE:

Kindly replace the Abstract of the Disclosure with the following text:

al

-- A computer-based method for generating a Downloadable ID to identify a Downloadable, including obtaining a Downloadable that includes one or more references to software components required by the Downloadable, fetching at least one software component identified by the one or more references, and performing a function on the Downloadable and the fetched software components to generate a Downloadable ID. A system and a computer-readable storage medium are also described and claimed. --

IN THE CLAIMS:

Kindly cancel claims 9 and 19 without prejudice.

Please substitute the following claims for the pending claims with the same number:

1. (Currently amended) A computer-based method for generating a Downloadable ID to identify a Downloadable, comprising [the steps of]:

obtaining a Downloadable that includes one or more references to software components required by the Downloadable;

fetching[, if the Downloadable includes one or more references to a component,] at least one software component identified by the one or more references; and

performing a function on the Downloadable and [all] the fetched software components [fetched] to generate a Downloadable ID.

2. (Original) The method of claim 1, wherein the Downloadable includes an applet.

A2

3. (Currently amended) The method of claim 1, wherein the Downloadable includes an [ActiveX^{TM}] active software control.

4. (Original) The method of claim 1, wherein the Downloadable includes a plugin.

5. (Original) The method of claim 1, wherein the Downloadable includes HTML code.

6. (Original) The method of claim 1, wherein the Downloadable includes an application program.

7. (Original) The method of claim 1, wherein the function includes a hashing function.

A

8. (Currently amended) The method of claim 1, wherein [the step of] said fetching includes [the step of] fetching the first software component referenced by the Downloadable.

9. (Cancelled)

10. (Currently amended) The method of claim 1, wherein [the step of] said fetching includes fetching all software components referenced by the Downloadable.

11. (Currently amended) A system for generating a Downloadable ID to identify a Downloadable, comprising:

a communications engine for obtaining a Downloadable that includes one or more references to software components required by the Downloadable; and

an ID generator coupled to the communications engine for fetching[*, if the Downloadable includes one or more references to a component,*] at least one software component identified by the one or more references, and for performing a function on the Downloadable and [all] the fetched software components [fetched] to generate a Downloadable ID.

12. (Original) The system of claim 11, wherein the Downloadable includes an applet.

13. (Currently amended) The system of claim 11, wherein the Downloadable includes an [ActiveX^{TM}] active software control.

14. (Original) The system of claim 11, wherein the Downloadable includes a plugin.

15. (Original) The system of claim 11, wherein the Downloadable includes HTML code.

A

16. (Original) The system of claim 11, wherein the Downloadable includes an application program.

17. (Original) The system of claim 11, wherein the function includes a hashing function.

18. (Currently amended) The system of claim 11, wherein the ID generator fetches the first software component referenced by the Downloadable.

19. (Cancelled)

20. (Currently amended) The method of claim 11, wherein the ID generator fetches all software components referenced by the Downloadable.

21. (Currently amended) A system for generating a Downloadable ID to identify a Downloadable, comprising:

means for obtaining a Downloadable that includes one or more references to software components required by the Downloadable;

means for fetching[, if the Downloadable includes one or more references to a component,] at least one software component identified by the one or more references; and

means for performing a function on the Downloadable and [all] the fetched software components [fetched] to generate a Downloadable ID.

22. (Currently amended) A computer-readable storage medium storing program code for causing a computer to perform the steps of:

obtaining a Downloadable that includes one or more references to software components required by the Downloadable;

fetching[, if the Downloadable includes one or more references to a component,] at least one software component identified by the one or more references; and

performing a function on the Downloadable and [all] the fetched software components [fetched] to generate a Downloadable ID.

A

REMARKS

Applicant has carefully studied the outstanding Office Action. The present amendment is intended to place the application in condition for allowance and is believed to overcome all of the objections and rejections made by the Examiner. Favorable reconsideration and allowance of the application are respectfully requested.

Applicant has canceled claims 9 and 19, and amended claims 1, 3, 8, 10, 11, 13, 18 and 20 - 22 to more properly claim the present invention. No new matter has been added. Claims 1 – 8, 10 – 18 and 20 – 22 are presented for examination.

Applicant notes that the page headers of the Office Action indicate an incorrect Application/Control Number.

In paragraphs 2 and 3 of the Office Action, the Examiner has objected to the abstract of the disclosure. Accordingly, applicant has amended the abstract so as to conform to the proper language and format.

In paragraphs 4 and 5 of the Office Action, the Examiner has rejected claims 1, 11, 21 and 22 under the judicially created doctrine of double patenting. Accordingly, applicant is submitting a terminal disclaimer with the present amendment.

In paragraphs 6 and 7 of the Office Action, the Examiner has rejected claims 3 and 13 under 35 U.S.C. §112, second paragraph as being indefinite. Applicant has amended these claims accordingly.

In paragraphs 8 and 9 of the Office Action, the Examiner has rejected claims 1, 7, 8, 10, 11, 17, 18, and 20 – 22 under 35 U.S.C. §102(e) as being anticipated by Apperson et al., U.S. Patent No. 5,978,484 ("Apperson").

In paragraphs 10 and 11 of the Office Action, the Examiner has rejected claims 2 – 4 and 12 – 14 under 35 U.S.C. §103(a) as being unpatentable over Apperson in view of Khare, "*Microsoft Authenticode Analyzed*", July 22, 1996, xent.com/FoRK-archive/summer96/0338.html, pg. 1 and 2 ("Khare").

In paragraph 12 of the Office Action, the Examiner has rejected claims 5, 6, 9, 15, 16 and 19 under 35 U.S.C. §103(a) as being unpatentable over Apperson. Applicant has canceled claims 9 and 19 without acquiescence to the Examiner's reasons for rejection and respectfully submits that rejection of those claims is thus rendered moot.

A

Distinctions between Claimed Invention and U.S. Patent No. 5,978,484 to Apperson et al in view of Khare, "Microsoft Authenticode Analyzed", July 22, 1996, xent.com/FoRK-archive/summer96/0338.html, pg. 1 and 2

The present invention concerns generation of an ID for mobile code downloaded to a client computer, referred to as a Downloadable. Specifically, the present invention fetches software components required by the Downloadable, and performs a hashing function on the Downloadable together with its fetched components (original specification / page 3, lines 11 – 14; page 15, lines 21 – 24; page 19, line 21 – page 20, line 6; FIG. 8). Thus, for a Java applet, the present invention fetches Java classes identified by the applet bytecode, and generates the Downloadable ID from the applet and the fetched Java classes; and for an ActiveX^{TM} control, the present invention fetches components listed in its .INF file, and generates a Downloadable ID from the ActiveX^{TM} control and the fetched components (original specification / page 9, lines 15 – 18).

An advantage of the present invention is that it produces the same ID for a Downloadable, regardless of which software components are included with the Downloadable and which software components are only referenced (original specification / page 9, lines 18 – 20; page 20, lines 5 and 6). The same Downloadable may be delivered with some required software components included and others missing, and in each case the generated Downloadable ID will be the same. Thus the same Downloadable is recognized through many equivalent guises.

Apperson describes use of digital certificates to authorize privileges for executable code, such as file I/O privileges, network privileges and registry privileges (Apperson / col. 2, lines 41 – 53; col. 4, lines 33 – 43; FIG. 2).

Khare describes Microsoft Corporation's implementation of digital signatures, referred to as Authenticode, as applied to ActiveX controls and Java applets.

In distinction to the present invention, Apperson and Khare do not teach fetching software components of executable code. In order to further clarify this distinction, applicant has amended the claims so as to refer to software components required by the Downloadable.

In paragraph 9 of the Office Action, the Examiner has indicated that Apperson discloses fetching components of a Downloadable. Applicant respectfully submits that Apperson's privilege request code does not include components of a Downloadable, but instead includes a list of "*privileges or*

A

privilege categories that the executable code might perform on the client machine" (Apperson / col. 2, lines 45 – 47).

The rejections of claims 1 –8 and 10 in paragraphs 8 - 12 of the Office Action will now be dealt with specifically.

As to amended independent method claim 1, applicant respectfully submits that the limitation in claim 1 of:

"*fetching at least one software component identified by the one or more references*"

is neither shown nor suggested in Apperson or Khare.

Because claims 2 – 8 and 10 depend from claim 1 and include additional features, applicant respectfully submits that claims 2 - 8 and 10 are not anticipated or rendered obvious by Apperson and Khare, taken alone or in combination.

Accordingly claims 1 – 8 and 10 are deemed to be allowable.

As to amended independent system claim 11, applicant respectfully submits that the limitation in claim 11 of:

"*an ID generator coupled to the communications engine for fetching at least one software component identified by the one or more references*"

is neither shown nor suggested in Apperson or Khare.

Because claims 12 – 18 and 20 depend from claim 11 and include additional features, applicant respectfully submits that claims 12 – 18 and 20 are not anticipated or rendered obvious by Apperson and Khare, taken alone or in combination.

Accordingly claims 12 – 18 and 20 are deemed to be allowable.

As to amended independent system claim 21, applicant respectfully submits that the limitation in claim 21 of:

"*means for fetching at least one software component identified by the one or more references*"

is neither shown nor suggested in Apperson or Khare.

Accordingly claim 21 is deemed to be allowable.

As to amended independent system claim 22, applicant respectfully submits that the limitation in claim 22 of:

"*fetching at least one software component identified by the one or more references*"

is neither shown nor suggested in Apperson or Khare.

Accordingly claim 22 is deemed to be allowable.

Support for Amended Claims in Original Specification

Regarding amended claims 1, 8, 10, 11, 18 and 20 - 22, fetching software components is described in the original specification on page 9, lines 13 – 18 and FIG. 8.

For the foregoing reasons, applicant respectfully submits that the applicable objections and rejections have been overcome and that the claims are in condition for allowance.

If the Examiner has any questions or needs any additional information, the Examiner is invited to telephone the undersigned attorney at (650) 843-3392. If for any reason an insufficient fee has been paid, please charge the insufficiency to Deposit Account No. 05-0150.

Date: July 31, 2003

Respectfully submitted,

Squire, Sanders & Dempsey L.L.P.
600 Hansen Way
Palo Alto, CA 94304-1043
Telephone: (650) 856-6500
Facsimile: (650) 843-8777

By:
Marc A. Sockol
Attorney for Applicant
Registration No. 40,823

A

EXHIBIT 2

09539667 03/30/00

713	181
Class	Subclass
ISSUE CLASSIFICATION	

C

PATENT NUMBER
6804780

5√

U.S. UTILITY Patent Application

O.I.P.E.	PATENT DATE
SCANNED JMO Q.A. MC	OCT 12 2004

APPLICATION NO.	CONT/PRIOR	CLASS	SUBCLASS	ART UNIT	EXAMINER
09/539667	1D	713	181	~~2785~~ 2151 ~~2151~~ ~~2184~~	Revak

APPLICANTS: Shlomo Touboul

TITLE: System and method for protecting a computer and a network from hostile downloadables

PTO-2040 12/99

ISSUING CLASSIFICATION

ORIGINAL		CROSS REFERENCE(S)						
CLASS	SUBCLASS	CLASS	SUBCLASS (ONE SUBCLASS PER BLOCK)					
713	181	713	201	176				
INTERNATIONAL CLASSIFICATION		717	178					
H04L	9/00							
G06F	11/30							

□ Continued on Issue Slip Inside File Jacket

9/10/04 Formal Drawings (10 shts) set / 3/30/00

☒ TERMINAL DISCLAIMER	DRAWINGS			CLAIMS ALLOWED	
	Sheets Drwg.	Figs. Drwg.	Print Fig.	Total Claims	Print Claim for O.G.
	10	10	8	18	1

□ The term of this patent subsequent to ____________ (date) has been disclaimed.

☒ The term of this patent shall not extend beyond the expiration date of U.S Patent. No. 6,092,194

□ The terminal ___ months of this patent have been disclaimed.

Christopher Revak 5/13/04
(Assistant Examiner) (Date)

AYAZ SHEIKH
SUPERVISORY PATENT EXAMINER
TECHNOLOGY CENTER 2100
5/24/04
(Primary Examiner) (Date)

Brenda Harris 7/04
(Legal Instruments Examiner) (Date)

NOTICE OF ALLOWANCE MAILED
06-4-04

ISSUE FEE

Amount Due	Date Paid
$ 665.00	9/3/04

ISSUE BATCH NUMBER

WARNING:
The information disclosed herein may be restricted. Unauthorized disclosure may be prohibited by the United States Code Title 35, Sections 122, 181 and 368. Possession outside the U.S. Patent & Trademark Office is restricted to authorized employees and contractors only.

Form PTO-436A
(Rev. 6/99)

FILED WITH: □ DISK (CRF) □ FICHE □ CD-ROM
(Attached in pocket on right inside flap)

(FACE)

Joint Trial Exhibit
JTX-52
Case No. 06-369 GMS

Attorney's Docket No.: 43426.00011

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

#11 SC
3/5/04

CERTIFICATE OF MAILING

I hereby certify that this paper (along with any paper referred to as being attached or enclosed) is being deposited with the United States Postal Service on the date shown below with sufficient postage as first class mail in an envelope addressed to Mail Stop RCE, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450, on

Date: 2/25/2004 By: Sandy Yi
Sandy Yi

In Re Patent Application of:

Shlomo Touboul

Application No: 09/539,667

Filed: March 30, 2000

For: SYSTEM AND METHOD FOR PROTECTING A COMPUTER AND A NETWORK FROM HOSTILE DOWNLOADABLES

Examiner: Christopher A. Revak

Art Unit: 2131

Mail Stop RCE
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

AMENDMENT AND RESPONSE WITH RCE

Sir:

In response to the Office Action dated October 27, 2003, the shortened statutory deadline for response ending on February 27, 2004 with the enclosed request for one-month extension and RCE, applicant respectfully requests that the above-identified application be amended as follows:

K

Attorney's Docket No.: 43426.00011

IN THE CLAIMS:

Please substitute the following claims for the pending claims with the same number:

B1

1. (Currently amended) A computer-based method for generating a Downloadable ID to identify a Downloadable, comprising:

obtaining a Downloadable that includes one or more references to software components required to be executed by the Downloadable;

fetching at least one software component identified by the one or more references; and

performing a hashing function on the Downloadable and the fetched software components to generate a Downloadable ID.

2. (Original) The method of claim 1, wherein the Downloadable includes an applet.

3. (Previously amended) The method of claim 1, wherein the Downloadable includes an active software control.

4. (Original) The method of claim 1, wherein the Downloadable includes a plugin.

5. (Original) The method of claim 1, wherein the Downloadable includes HTML code.

6. (Original) The method of claim 1, wherein the Downloadable includes an application program.

~~7. (Original) The method of claim 1, wherein the function includes a hashing function.~~

C

~~8.~~ 7 (Previously amended) The method of claim 1, wherein said fetching includes fetching ~~the~~ a first software component referenced by the Downloadable.

9. (Cancelled)

8 ~~10~~. (Previously amended) The method of claim 1, wherein said fetching includes fetching all software components referenced by the Downloadable.

9 ~~11~~. (Currently amended) A system for generating a Downloadable ID to identify a Downloadable, comprising:

a communications engine for obtaining a Downloadable that includes one or more references to software components required to be executed by the Downloadable; and

an ID generator coupled to the communications engine ~~for fetching~~ that fetches at least one software component identified by the one or more references, and for performing a hashing function on the Downloadable and the fetched software components to generate a Downloadable ID.

10 ~~12~~. (Original) The system of claim 9 ~~11~~, wherein the Downloadable includes an applet.

11 ~~13~~. (Previously amended) The system of claim 9 ~~11~~, wherein the Downloadable includes an active software control.

12 ~~14~~. (Original) The system of claim 9 ~~11~~, wherein the Downloadable includes a plugin.

13 ~~15~~. (Original) The system of claim 9 ~~11~~, wherein the Downloadable includes HTML code.

14 ~~16~~. (Original) The system of claim 9 ~~11~~, wherein the Downloadable includes an application program.

~~17. (Original) The system of claim 11, wherein the function includes a hashing function.~~ C

15 ~~18~~. (Previously amended) The system of claim 9 ~~11~~, wherein the ID generator fetches ~~the~~ a first software component referenced by the Downloadable.

25

19. (Cancelled)

16 ~~20~~. (Previously amended) The method of claim 9 ~~11~~, wherein the ID generator fetches all software components referenced by the Downloadable.

17 ~~21~~. (Currently amended) A system for generating a Downloadable ID to identify a Downloadable, comprising:

means for obtaining a Downloadable that includes one or more references to software components required to be executed by the Downloadable;

means for fetching at least one software component identified by the one or more references; and

means for performing a hashing function on the Downloadable and the fetched software components to generate a Downloadable ID.

18 ~~22~~. (Currently amended) A computer-readable storage medium storing program code for causing a computer to perform the steps of:

obtaining a Downloadable that includes one or more references to software components required to be executed by the Downloadable;

fetching at least one software component identified by the one or more references; and

performing a hashing function on the Downloadable and the fetched software components to generate a Downloadable ID.

REMARKS

Claims 1 – 8, 10 – 18 and 20 – 22 are presented for examination. Claims 1, 11, 21 and 22 are being amended. Applicant respectfully requests reconsideration of the application in view of the amendments above and remarks below.

Applicant would like to thank the Examiner for the interview on January 27, 2004 to discuss the office action, the Apperson reference and the current claim set. During the interview, Applicant and the Examiner discussed how the system described in the Apperson reference associates privileges to a Downloadable and then allows the Downloadable to execute only those operations allowed by the associated privileges. Applicant and the Examiner discussed how the privileges in Apperson are monitored by the browser, not executed by the Downloadable, and further how the Apperson reference does not generate Downloadable IDs based on the fetched executable components. Further, Applicant and the Examiner discussed adding the language "to be executed" into the claim language to further show that the additional components are "to be executed," thereby highlighting that difference between the Apperson reference and claimed invention.

Specifically, in paragraphs 1 and 2 of the office action, the Examiner rejected claims 1, 5-8, 10, 11, 15-18 and 20-22 under 35 USC § 103(a) over Apperson. Apperson describes the use of digital certificates to authorize privileges for executable code. Such privileges include file I/O privileges, network privileges and registry privileges (Apperson / col. 2, lines 41 – 53; col. 4, lines 33 – 43; FIG. 2).

Apperson, however, does not teach fetching at least one software component referenced by a Downloadable, where the software component is "required to be executed by the Downloadable" and "performing a function on the Downloadable and the fetched software components to generate a Downloadable ID" as recited in independent claims 1, 11, 21 and 22, as amended. As will be recognized by those skilled in the art, in some embodiments, the Downloadable ID may be used to recognize the "same" Downloadable regardless of how the Downloadable is subdivided and/or downloaded before and/or during execution. Since all other claims depend from these independent claims, Applicant respectfully submits that they are distinguishable over Apperson for at least the same reasons.

In paragraph 3, the Examiner rejected claims 2-4 and 12-14 over Apperson in view of Khare. Khare describes Microsoft Corporation's implementation of digital signatures, referred to as Authenticode, as applied to ActiveX controls and Java applets. Like Apperson, Khare does not teach fetching at least one software component referenced by a Downloadable, where the software component is "required to be executed by the Downloadable" and "performing a function on the Downloadable and the fetched software components to generate a Downloadable ID" as recited in independent claims 1, 11, 21 and 22, as amended. Since claims 2-4 and 12-14 depend from claims 1 and 11, respectively, Applicant respectfully submits that they are patentable for at least the same reasons.

For the foregoing reasons, applicant respectfully submits that the claims are in condition for allowance.

If the Examiner has any questions or needs any additional information, the Examiner is invited to telephone the undersigned attorney at (650) 843-3392. If for any reason an insufficient fee has been paid, please charge the insufficiency to Deposit Account No. 05-0150.

Date: February 25, 2004

Respectfully submitted,

Squire, Sanders & Dempsey L.L.P.
600 Hansen Way
Palo Alto, CA 94304-1043
Telephone: (650) 856-6500
Facsimile: (650) 843-8777

By:
Marc A. Sockol
Attorney for Applicant
Registration No. 40,823

EXHIBIT 3

Seidl, Christopher A.

From: Kobialka, Lisa (Perkins Coie) [LKobialka@perkinscoie.com]
Sent: Saturday, January 05, 2008 1:57 PM
To: Seidl, Christopher A.; Holdreith, Jake M.; Farnan, Kelly E.; Foster, Trevor J.; Moravetz, Amy
Cc: Rovner, Philip A.
Subject: RE: Special Verdict Form

No.

Lisa Kobialka
Perkins Coie LLP
101 Jefferson Drive
Menlo Park, CA 94025
Ph: (650) 838-4447
Fax: (650) 838-4350

From: Seidl, Christopher A. [mailto:CASeidl@rkmc.com]
Sent: Thursday, January 03, 2008 6:59 PM
To: Kobialka, Lisa (Perkins Coie); Holdreith, Jake M.; Farnan, Kelly E.; Foster, Trevor J.; Moravetz, Amy
Subject: RE: Special Verdict Form

>>>> Please read the confidentiality statement below <<<<
Lisa,

That is correct: Is Finjan asserting contributory infringement or inducement on Finjan's patents at trial?

Chris

From: Kobialka, Lisa (Perkins Coie) [mailto:LKobialka@perkinscoie.com]
Sent: Thursday, January 03, 2008 8:34 PM
To: Seidl, Christopher A.; Holdreith, Jake M.; Farnan, Kelly E.; Foster, Trevor J.; Moravetz, Amy
Cc: Rovner, Philip A.; Andre, Paul (Perkins Coie); Hannah, James (Perkins Coie); Wharton, Meghan (Perkins Coie); Kastens, Kris (Perkins Coie); Dennison, Steven (Perkins Coie)
Subject: Special Verdict Form

Chris,

Attached please find Finjan's Special Verdict Form.

When we spoke today, you asked whether we intended to pursue our indirect infringement claims. I told you I did not know. Can you tell me what you are referring to because I am afraid that misunderstood your question. Are you asking whether we are asserting contributory or inducing infringement on Finjan's patents or asking something else?

Lisa

<<13831763_1.DOC>>

NOTICE: This communication may contain privileged or other confidential information. If you have received it in error, please advise the sender by reply email and immediately delete the message and any attachments without copying or disclosing the contents. Thank you.

Information contained in this e-mail transmission may be privileged, confidential and covered by the Electronic Communications Privacy Act, 18 U.S.C. Sections 2510-2521.

If you are not the intended recipient, do not read, distribute, or reproduce this transmission.

If you have received this e-mail transmission in error, please notify us immediately of the error by return email and please delete the message from your system.

Pursuant to requirements related to practice before the U. S. Internal Revenue Service, any tax advice contained in this communication (including any attachments) is not intended to be used, and cannot be used, for purposes of (i) avoiding penalties imposed under the U. S. Internal Revenue Code or (ii) promoting, marketing or recommending to another person any tax-related matter.

Thank you in advance for your cooperation.

Robins, Kaplan, Miller & Ciresi L.L.P.
http://www.rkmc.com

EXHIBIT 4

Figure 2
Correction of Royalty Base for Webwasher Software
(Starting from Mr. Parr's)

	CyberGuard					Secure Computing							
	(a)			(b)		(c)	(d)						
	2004 Q4	2005 Q1	2005 Q2	2005 Q3	2005 Q4	2006 Q1	2005 Q2	2005 Q3	2006 Q4	2007 Q1	2007 Q2	2007 Q3	TOTAL
Parr royalty base "Webwasher Software"[1]	$4,003,225	$3,464,440	$4,568,859	$3,392,527	$3,234,959	$2,455,242	$4,825,283	$5,350,016	$6,915,461	$4,813,226	$5,263,262	$2,591,843	$50,878,343
Parr royalty base after correction for data entry errors[2]	$4,003,225	$3,464,440	$4,568,859	$3,392,527	$3,234,959	$2,455,242	$4,825,283	$5,350,016	$6,915,520	$4,813,226	$5,243,262	$2,591,843	$50,858,402
Deduct maintenance/support/services/inspector revenues[3]	$372,089	$469,567	$483,071	$67,684	$45,513								
Royalty Base after service deductions[4]	$3,631,136	$2,994,873	$4,085,788	$3,324,843	$3,189,446	$2,455,242	$4,825,283	$5,350,016	$6,915,520	$4,813,226	$5,243,262	$2,591,843	$49,420,478
Deduct non-U.S. software revenues[5]	$2,749,718	$2,347,797	$3,356,154	$2,623,941	$2,517,087								
Royalty Base after non-U.S. software deduction[6]	$881,418	$647,076	$729,634	$700,902	$672,359	$2,455,242	$4,825,283	$5,350,016	$6,915,520	$4,813,226	$5,243,262	$2,591,843	$35,825,781
U.S. % of software sales[7]	24.3%	21.6%	17.9%	21.1%	21.1%								
Deduct non-proactive scan modules[8]	$561,399	$412,140	$464,724	$446,424	$428,244	$1,563,811	$3,145,400	$3,407,572	$4,528,075	$3,644,363	$3,744,484	$2,029,765	$24,376,402
Percent of revenue attributable to non-proactive scan modules[9]	63.7%	63.7%	63.7%	63.7%	63.7%	63.7%	65.2%	63.7%	65.5%	75.7%	71.4%	78.3%	
Royalty base after non-proactive scan deduction[10]	$320,019	$234,936	$264,910	$254,478	$244,115	$891,431	$1,679,883	$1,942,444	$2,387,445	$1,168,863	$1,498,777	$562,078	$11,448,380
Deduct proactive module sales to U.S. Federal government[11]						$949	$390,612	$0	$21,250	$0	$0	$34,933	
Royalty base after gov't deduction[12]	$320,019	$234,936	$264,910	$254,478	$244,115	$890,483	$1,289,271	$1,942,444	$2,366,195	$1,168,863	$1,498,777	$527,145	$11,001,636

Corrected Royalty Base: Webwasher Software $11,001,636

[1] Parr Report, Exhibit 3.
[2] It appears that the Parr Report, Exhibit 3 contains data entry errors in Q4 2006 and Q2 2007.
[3] Data in column (a) is from SC189211. Data in column (b) is from SC189214. Mr. Parr properly deducts these revenues in the post-2005 time period.
[4] Equals row 2 minus row 3.
[5] Data in column (a) is from SC189211. Data in column (b) equals (1 minus row 7) multiplied by row 4. I understand that after Secure Computing Corporation acquired CyberGuard in January 2006, non-U.S. sales should no longer be deducted.
[6] Equals row 4 minus row 5.
[7] For column (a), equals row 6 divided by row 4. For column (b), equals average of row 6, column (a).
[8] For columns (a), (b), and (c), equals row 6 multiplied by row 9. Data in column (d) is from SC189203.
[9] To be conservative, columns (a), (b), and (c) equal the minimum of row 9, column (d). For column (d), equals row 8 divided by row 6.
[10] Equals row 6 minus row 8.
[11] SC189649.
[12] Equals row 10 minus row 11.

Defendant's Trial Ex.
DTX - 1336
Case No. 06-369 GMS

EXHIBIT 5

Figure 2A
Correction of Royalty Base for Appliances
(Starting from Mr. Parr's)

	CyberGuard					Secure Computing							
	(a)			(b)		(c)	(d)						
	2004 Q4	2005 Q1	2005 Q2	2005 Q3	2005 Q4	2006 Q1	2006 Q2	2006 Q3	2006 Q4	2007 Q1	2007 Q2	2007 Q3	TOTAL
Parr royalty base "Appliance Sales"[1]			$8,431,199			$1,500,000	$2,722,814	$397,674	$317,322	$590,005	$866,550	$550,096	$15,375,660
Deduct CyberGuard TSP appliance sales[2]			$8,431,199			$1,500,000	$2,200,000						$12,131,199
Royalty Base after deducting TSP appliance revenues[3]			$0			$0	$522,814	$397,674	$317,322	$590,005	$866,550	$550,096	$3,244,461
Deduct Webwasher appliance sales to U.S. Federal Government[4]						$0	$406,934	$38,858	$12,953	$213,721	$25,906	$4,013	$702,385
Royalty Base after deducting Webwasher appliance sales to U.S. Federal Government[5]						$0	$115,880	$358,816	$304,369	$376,284	$840,645	$546,083	$2,542,076

Corrected Royalty Base: Webwasher Appliances $2,542,076

[1] Parr Report, Exhibit 3.
[2] Parr Report, Exhibit 3.
[3] Row 1 minus row 2.
[4] SC169649.
[5] Row 3 minus row 4.

Defendant's Trial Ex.
DTX - 1337
Case No. 06-369 GMS

EXHIBIT 6

US006571338B1

(12) **United States Patent**
Shaio et al.

(10) Patent No.: **US 6,571,338 B1**
(45) Date of Patent: ***May 27, 2003**

(54) MAINTAINING PACKET SECURITY IN A COMPUTER NETWORK

(75) Inventors: Sami Shaio, San Francisco, CA (US); Arthur Van Hoff, Mountain View, CA (US)

(73) Assignee: Sun Microsystems Inc., Mountain View, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 1347 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: 08/575,743

(22) Filed: Dec. 20, 1995

(51) Int. Cl.^{7} H04L 9/00
(52) U.S. Cl. 713/201; 713/153
(58) Field of Search 395/187.01, 188.01, 395/186, 609, 200.68, 200.69, 200.7, 200.72, 200.73, 200.74; 380/23, 25; 707/9, 10; 713/201, 153, 154, 160, 161

(56) References Cited

U.S. PATENT DOCUMENTS

5,113,499	A	*	5/1992	Ankney et al.	395/325
5,311,593	A	*	5/1994	Carmi	370/400
5,400,334	A	*	3/1995	Hayssen	370/85.4
5,414,694	A	*	5/1995	Crayford et al.	370/13.1
5,438,568	A	*	8/1995	Weisser, Jr.	370/389
5,530,758	A	*	6/1996	Marino, Jr. et al.	380/49
5,548,649	A	*	8/1996	Jacobson	380/49
5,550,984	A	*	8/1996	Gelb	395/187.01
5,559,883	A	*	9/1996	Williams	380/4
5,572,533	A	*	11/1996	Sunada et al.	371/20.1
5,572,643	A	*	11/1996	Judson	395/774
5,581,559	A	*	12/1996	Crayford et al.	395/186
5,590,285	A	*	12/1996	Krause et al.	395/200.2
5,615,340	A	*	3/1997	Dai et al.	395/200.17
5,623,600	A	*	4/1997	Ji et al.	395/187.01
5,623,601	A	*	4/1997	Vu	395/187.01
5,638,515	A	*	6/1997	Futral	395/200.11

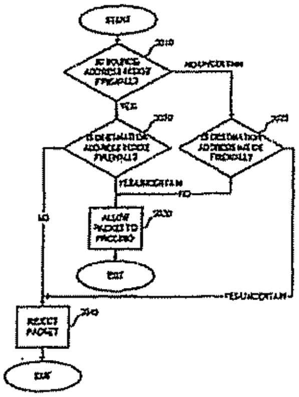
* cited by examiner

Primary Examiner—Scott Baderman
(74) *Attorney, Agent, or Firm*—Kang S. Lim, Esq.; James D. Ivey, Esq.

(57) ABSTRACT

The present invention provides a method and apparatus for determining the trust worthiness of executable packets, e.g., internet applets, being transmitted within a computer network. The computer network includes both secured computers and unsecured computers, which are associated with secured nodes and unsecured nodes, respectively. Each executable packet has a source address and a destination address. In one embodiment, an intelligent firewall determines within a first degree of certainty whether the source address of an executable packet arriving at one of the secured computers is associated with anyone of the secured nodes, and also determines within a second degree of certainty whether the destination address of the executable packet is associated with anyone of the secured nodes. If the firewall determines within the first degree of certainty that the source address is associated with anyone of the secured nodes, and further determines within the second degree of certainty or is uncertain whether the destination address is associated with anyone of the secured nodes, then the firewall permits the executable packet to execute on the secured computer. Alternatively, if the firewall determines within the first degree of certainty or is uncertain whether the source address is associated with anyone of the secured nodes, and further determines within the second degree of certainty that the destination address is not associated with anyone of the secured nodes, then the firewall also permits the executable packet to proceed to the secured computer.

20 Claims, 7 Drawing Sheets



Defendant's Trial Ex.
DTX - 1021
Case No. 06-369 GMS

DEPOSITION EXHIBIT
11
Wallach
PENGAD 800-631-6989

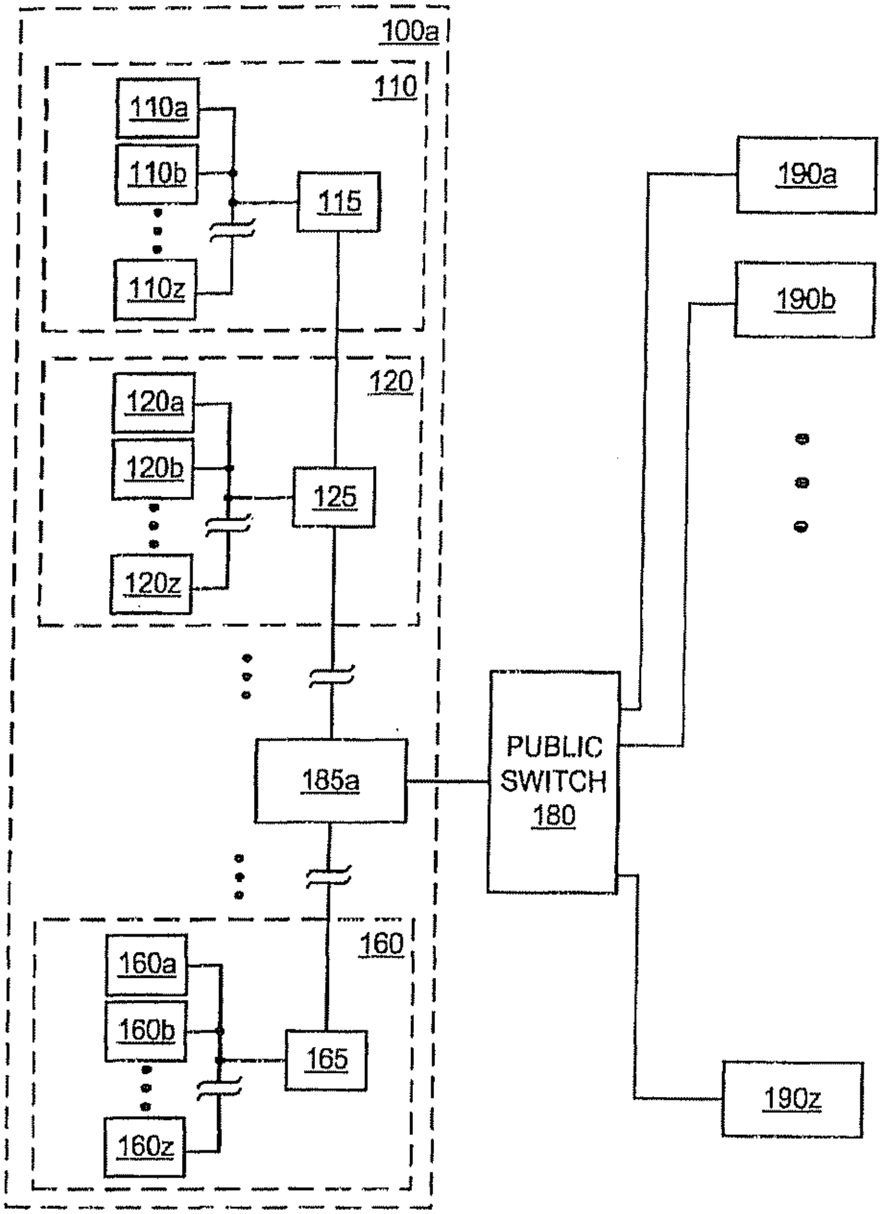


FIGURE 1A PRIOR ART

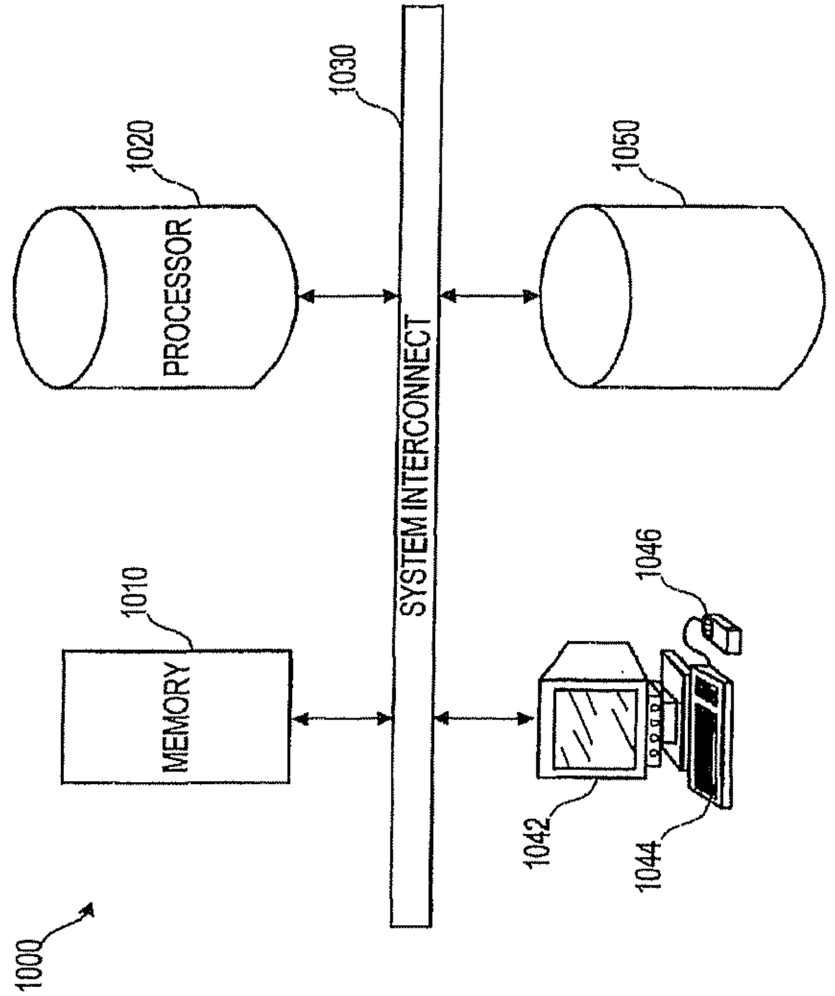


FIGURE 1B PRIOR ART

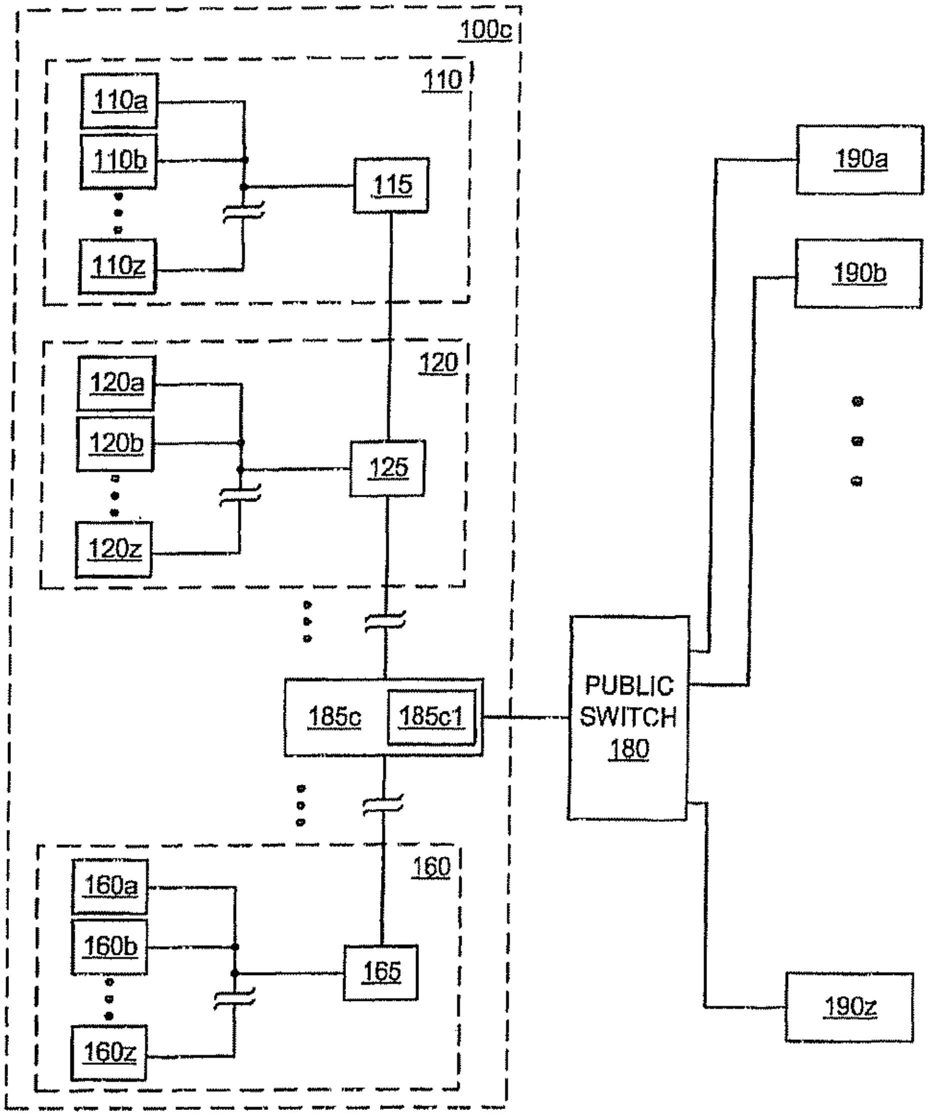

100c
110
110a
110b
110z
115
120
120a
120b
120z
125
185c
185c1
160
160a
160b
160z
165
PUBLIC SWITCH 180
190a
190b
190z

FIGURE 1C

		DESTINATION ADDRESS		
		INSIDE (I) FIREWALL	UNCERTAIN (U)	OUTSIDE (O) FIREWALL
SOURCE ADDRESS	INSIDE FIREWALL	✓	✓	?
	UNCERTAIN	?	?	✓
	OUTSIDE FIREWALL	?	?	✓

FIGURE 2A

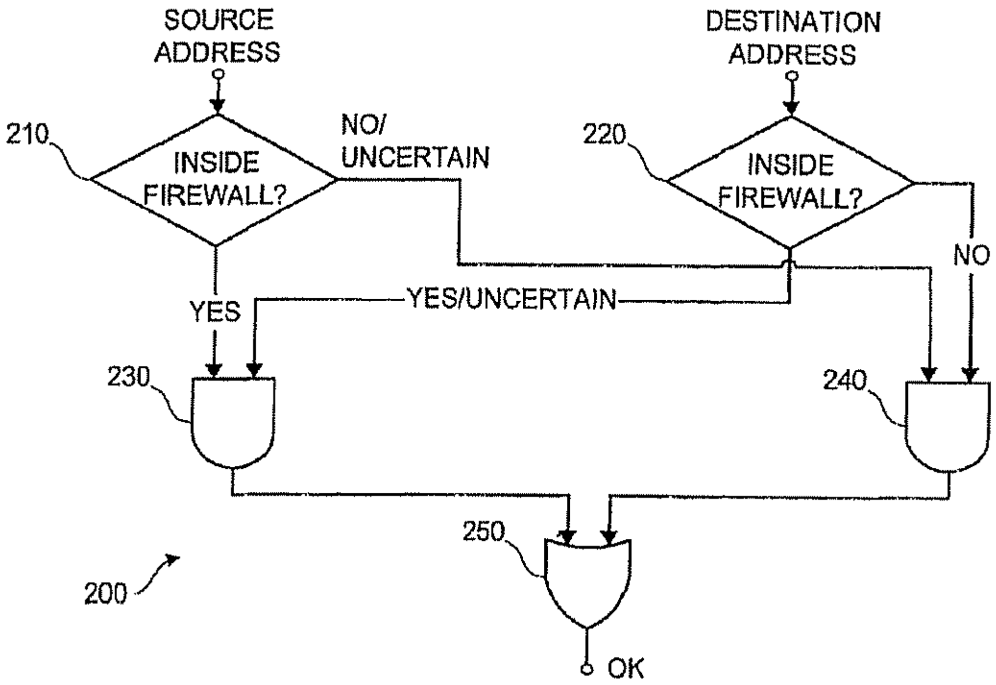


FIGURE 2B

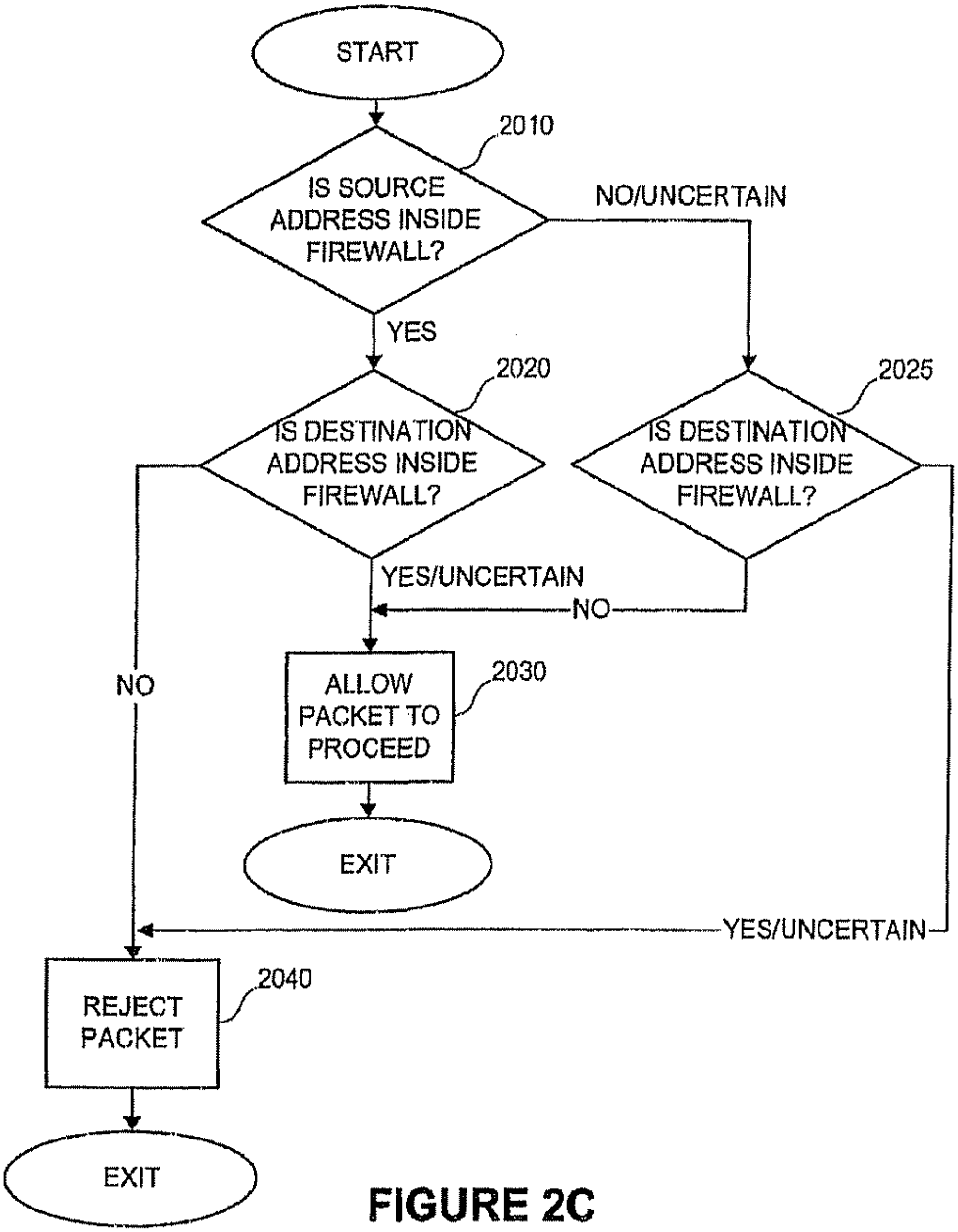

START
2010
IS SOURCE ADDRESS INSIDE FIREWALL?
NO/UNCERTAIN
YES
2020
IS DESTINATION ADDRESS INSIDE FIREWALL?
2025
IS DESTINATION ADDRESS INSIDE FIREWALL?
YES/UNCERTAIN
NO
NO
2030
ALLOW PACKET TO PROCEED
EXIT
YES/UNCERTAIN
2040
REJECT PACKET
EXIT

FIGURE 2C

		DESTINATION ADDRESS		
		INSIDE (I) FIREWALL	UNCERTAIN (U)	OUTSIDE (O) FIREWALL
SOURCE ADDRESS	INSIDE FIREWALL	✓	✓	✓
	UNCERTAIN	X	X	✓
	OUTSIDE FIREWALL	X	X	✓

FIGURE 3A

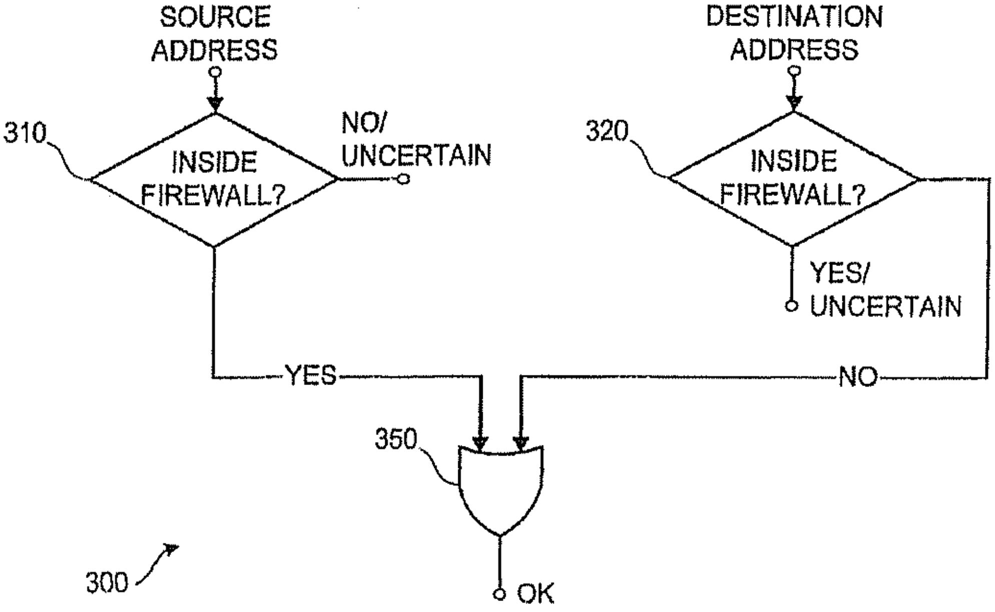


FIGURE 3B

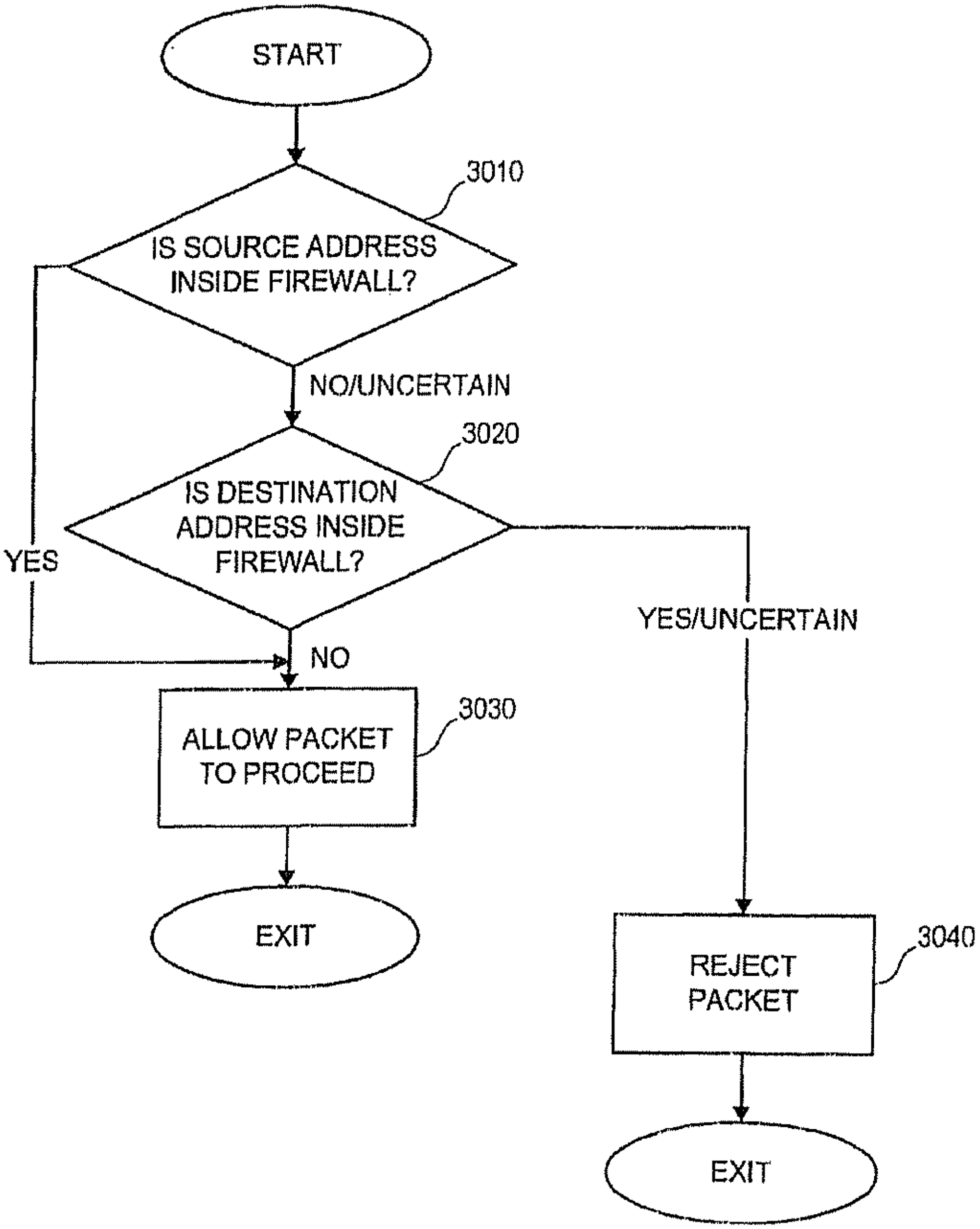

START
3010
IS SOURCE ADDRESS INSIDE FIREWALL?
NO/UNCERTAIN
3020
IS DESTINATION ADDRESS INSIDE FIREWALL?
YES
YES/UNCERTAIN
NO
3030
ALLOW PACKET TO PROCEED
EXIT
3040
REJECT PACKET
EXIT

FIGURE 3C

MAINTAINING PACKET SECURITY IN A COMPUTER NETWORK

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to the field of security in a computer network. More particularly, the present invention relates to the field of packet security in a wide area network (WAN). An example of a byte code verifier system that can be used in connection with the present invention is disclosed in the following copending patent application, which is incorporated herein by reference: "B YTECODE PROGRAM INTERPRETER APPARATUS AND METHOD WITH PREVERIFICATION OF DATA TYPE RESTRICTIONS AND OBJECT INITIALIZATION", Ser. No. 08/575,291, by Frank Yellin and James Gosling, filed on the same day as the present application.

2. Description of the Related Art

FIG. 1A illustrates a typical computing environment wherein clusters of secured computers **110***a*, **110***b*, . . . **100***z*, **120***a*, **120***b*, . . . **120***z*, . . . **160***a*, **160***b*, . . . **160***z* are coupled to each other to form local area networks (LANs) **110**, **120**, . . . **160**, respectively. Exemplary technologies employed for interconnecting LANs include Ethernet and Token-ring. In turn, LANs **110**, **120**, . . . **160** can be coupled to each other via network nodes **115**, **125**, . . . , **165** to form a secured wide area network (SWAN) **100***a*. Typical SWAN links include dedicated leased lines and satellite links which are less vulnerable to attack than public networks in general.

In most commercial computing implementations, security is maintained by identifying internal computers whose use can be closely monitored, e.g., secured computers **110***a*, **110***b*, . . . **110***z*, **120***a*, **120***b*, . . . **120***z*, . . . **160***a*, **160***b*, . . . **160***z*, and also by enforcing a strict policy of not allowing any new executable programs to be executed in any one of the secured computers until these new programs have been verified as virus-free. Viruses can cause a variety of problems such as damage to hardware, software, and/or data, release information to unauthorized personnel, and/or cause a host computer to become unusable through resource depletion.

Unfortunately, most commercial networks have a need to be connected to external unsecured computers, such as the computers of telecommuting-employees and customers. For example, SWAN **100***a* may be coupled to external unsecured computers **190***a*, **190***b*, . . . **190***z* via an externally-accessible node **185***a* and a public switch **180**.

As this need to connect SWAN **100***a* to an increasing number of unsecured computers **190***a*, **190***b*, **190***z* via public switch **180** grows, the problem of guarding the secured computers of SWAN **100***a* against unauthorized data access and/or data corruption becomes increasing difficult. This problem is compounded by the proliferation of computers coupled to publicly and freely accessible WANs such as the Internet. Hence, externally accessible node **185***a*, the weakest point of the otherwise-secure SWAN **100***a*, is increasingly vulnerable to hackers.

Several techniques have been developed to minimize the vulnerability of node **185***a* to any uninvited intrusion. For example as discussed above, whenever possible, dedicated trunk lines of switch **180** are used to connect node **185***a* to unsecured computers **190***a*, **190***b*, . . . **190***z*. A less costly but less secure alternative is the enforcement of a dialback protocol over a public network, in which an unsecured

computer, e.g., computer **190***a*, dials up node **185***a*, and then identifies the remote user's identity and location before hanging up. Subsequently, node **185***a* dials back computer **190***a* at its pre-authorized location using a pre-authorized telephone number to ensure that the remote user is indeed located at the preauthorized location.

Additional security at the packet level can also be provided at node **185***a*, wherein node **185***a* functions as a dumb "firewall" which allows only pure ASCII files, e.g., textual emails, and prohibits all attachments of the emails from leaving and/or entering SWAN **100***a*. Alternatively, node **185***a* may scan all incoming packets to identify and prevent any untested executable code from entering SWAN **100***a*.

Although the above-described security measures work fairly well for the exchange of data packets between SWAN **100***a* and unsecured computers **190***a*, **190***b*, . . . **190***z*, they are too cumbersome and/or inadequate for exchanging packets which include executable code. For example, in receiving an executable Internet application based on Hot Java, a programming language that supports executable applets, such a broad prohibition of executable code will effectively prevent any untested Hot Java applets from being received and subsequently executed.

Hence, there is a need for an intelligent firewall that provides real-time security testing of network packets, which may include executable code such as applets, and determines the risk level, i.e, trust worthiness, of each packet before permitting a lower-risk subset of the network packets to execute on anyone of the secured computers of SWAN **100***a* in a manner transparent to a user.

SUMMARY OF THE INVENTION

The present invention provides a method and apparatus for determining the trust worthiness of executable packets, e.g., internet applets, being transmitted within a computer network. The computer network includes both secured computers and unsecured computers, which are associated with secured nodes and unsecured nodes, respectively. Each executable packet has a source address and a destination address.

In one embodiment, an intelligent firewall determines within a first degree of certainty whether the source address of an executable packet arriving at one of the secured computers is associated with anyone of the secured nodes, and also determines within a second degree of certainty whether the destination address of the executable packet is associated with anyone of the secured nodes.

If the firewall determines within the first degree of certainty that the source address is associated with anyone of the secured nodes, and further determines within the second degree of certainty or is uncertain whether the destination address is associated with anyone of the secured nodes, then the firewall permits the executable packet to proceed to the secured computer.

Alternatively, if the firewall determines within the first degree of certainty or is uncertain whether the source address is associated with anyone of the secured nodes, and further determines within the second degree of certainty that the destination address is not associated with anyone of the secured nodes, then the firewall also permits the executable packet to proceed to the secured computer.

In another embodiment, the intelligent firewall determines within the first degree of certainty whether the source address of an executable packet arriving at one of the secured computers is associated with anyone of the secured nodes, or determines within the second degree of certainty whether the destination address of the executable packet is associated with anyone of the secured nodes.

If the firewall determines within the first degree of certainty that the source address is associated with anyone of the secured nodes, then the firewall permits the executable packet to proceed to the secured computer. Alternatively, the firewall determines within the second degree of certainty whether the destination address of the executable packet is associated with anyone of the secured nodes, then the firewall also permits the executable packet to proceed to the secured computer.

In the above-described embodiments, if none of the above-described trust-worthiness conditions are satisfied, then the firewall rejects the executable packet, thereby minimizing the risk of damage to the secured computer.

DESCRIPTION OF THE DRAWINGS

The objects, features and advantages of the system of the present invention will be apparent from the following description in which:

FIG. **1A** is a block diagram of a typical computer network.

FIG. **1B** is a block diagram of a general purpose computer system.

FIG. **1C** is a block diagram of a computer network of the present invention.

FIGS. **2A**, **2B** and **2C** are a truth table, a block diagram and a flowchart, respectively, illustrating one embodiment of the intelligent firewall of the present invention.

FIGS. **3A**, **3B** and **3C** are a truth table, a block diagram and a flowchart, respectively, illustrating another embodiment of the intelligent firewall of the present invention

DESCRIPTION OF THE PREFERRED EMBODIMENT

In the following description, numerous details provide a thorough understanding of the invention. These details include functional blocks and exemplary algorithms to assist one in implementing an intelligent network firewall. In addition, while the present invention is described with reference to a specific computer network architecture and firewall algorithms for protecting the network, the invention is applicable to a wide range of network architectures and environments. In other instances, well-known circuits and structures are not described in detail so as not to obscure the invention unnecessarily.

FIG. **1C** illustrates a secured wide area network (SWAN) **100c** of the present invention, which includes clusters of secured computers **110a**, **110b**, . . . **110z**, **120a**, **120b**, . . . **120z**, . . . **160a**, **160b**, . . . **160z**, coupled to each other to form local area networks (LANs) **110**, **120**, . . . **160**, respectively. LANs **110**, **120**, . . . **160** can be coupled to each other via network nodes **115**, **125**, . . . , **165**. SWAN **100c** is coupled to external unsecured computers **190a**, **190b**, . . . **190z** via an externally-accessible network node **185c** and a public switch **180**.

In accordance with the present invention, node **185c** includes an intelligent firewall **185c1**. Node **185c** can be the general purpose computer **1000** of FIG. **1B** or a dedicated network packet router (not shown) suitable for implementing firewall **185c1**. For the purpose of illustrating the following examples, "outside firewall **185c1**" is equivalent to outside secured wide area network (SWAN) **100c**.

FIGS. **2A**, **2B** and **2C** are a truth table, a block diagram and a flowchart, respectively, illustrating the operation of one embodiment of intelligent firewall **185c1**. Appendix A is an exemplary pseudo-code implementation of this embodiment.

Referring to the flowchart of FIG. **2C**, when firewall **185c1** receives an incoming or an outgoing network packet, an examination of the source address of the network packet is performed (step **2010**).

If firewall **185c1** determines within a degree of certainty that the source address identifies the packet as originating from one of the secured computer systems within SWAN **100c**, and upon examination of the destination address of the packet (step **2020**), firewall **185c1** is uncertain or determines that the destination address of the packet is inside SWAN **100c**, then the packet is allowed to proceed (step **2030**). Alternatively, if firewall **185c1** is either uncertain or determines that the source address is outside SWAN **100c**, and upon examination of the destination address of the packet (step **2025**), firewall **185c1** determines within a degree of certainty that the destination address of the packet is outside SWAN **100c**, then the packet is also allowed to proceed (step **2030**).

Conversely, if firewall **185c1** is uncertain or determines that the source address of the packet is outside SWAN **100c**, and upon examination of the destination address (step **2025**), is uncertain or determines that the destination address of the packet is inside SWAN **100c**, then the packet is rejected, i.e., prevented from proceeding to anyone of the secured computers of SWAN **100c** (step **2040**). Similarly, if firewall **185c1** determines within a degree of certainty that the source address identifies the packet as originating from one of the secured computer systems inside SWAN **100c**, and upon examination of the destination address of the packet (step **2020**), determines that the destination address of the packet is outside SWAN **100c**, then the packet is also rejected (step **2040**).

In this embodiment, a source/destination network address is considered uncertain if there is no match between the network address and a list of pre-approved secured network addresses inside SWAN **100c**. Other definitions of uncertainty are possible. For example, network addresses may include a prefix field and a machine field, with the prefix field identifying clusters of computer systems coupled to the respective network nodes, and the machine field identifying computer systems within each cluster. Hence, even though firewall **185c1** may recognize the prefix field of the packet as one associated with a secured network node within SWAN **100c**, if the machine field of the same packet does not match one of the pre-approved identifiers, the result is a partial match and the network address of the packet is considered an uncertain address by firewall **185c1**

FIGS. **3A**, **3B** and **3C** are a truth table, a block diagram and a flowchart, respectively, illustrating the operation of another embodiment of intelligent firewall **185c1**. Referring to the flowchart of FIG. **3C**, when firewall **185c1** receives an incoming or an outgoing network packet, an examination of the source address of the network packet is performed (step **3010**).

If firewall **185c1** determines within a degree of certainty that the source address identifies the packet as originating from one of the secured computer systems inside SWAN **100c**, then the packet is allowed to proceed (step **3030**). Alternatively, if firewall **185c1** is either uncertain or determines that the source address of the packet is outside SWAN **100c**, and upon examination of the destination address of the packet (step **3020**), firewall **185c1** determines within a degree of certainty that the destination address of the packet is outside SWAN **100c**, then the packet is allowed to proceed (step **3030**).

Conversely, if firewall 185c1 is uncertain or determines that the source address of the packet is outside SWAN 100c, and upon examination of the destination address (step 3020), is uncertain or determines that the destination address of the packet is inside SWAN 100c, then the packet is rejected (step 3040).

Additional security may be provided by intelligent firewall 185c1. For example, a byte code verifier may parse the executable code portion of the packet to eliminate invalid and/or non-conforming instructions in an attempt to reduce the probability of viruses. An example of a byte code verifier system that can be used in connection with the present invention is disclosed in the above-mentioned copending patent application, entitled: "BYTECODE PROGRAM INTERPRETER APPARATUS AND METHOD WITH PRE-VERIFICATION OF DATA TYPE RESTRICTIONS AND OBJECT INITIALIZATION". Other modifications and additions are also possible without departing from the spirit of the invention. Accordingly, the scope of the invention should be limited by the following claims.

What is claimed is:

1. A method for determining the trust worthiness of executable packets in a computer network having a plurality of secured computers and a plurality of unsecured computers, each executable packet having a source address and a destination address, said method comprising the steps of:

a) determining within a first degree of certainty whether a source address of one said executable packet is associated with anyone of said plurality of secured computers, said source address is not associated with anyone of said plurality of secured computers, or association of said source address with anyone of said plurality of secured computers is uncertain; and

b) determining within a second degree of certainty whether a destination address of said one executable packet is associated with anyone of said plurality of secured computers, said destination address is not associated with anyone of said plurality of secured computers, or association of said destination address with anyone of said plurality of secured computers is uncertain.

2. The method of claim 1 wherein if the determining step a) determines within said first degree of certainty that said source address is associated with anyone of said plurality of secured computers and if the determining step b) determines within said second degree of certainty or is uncertain whether said destination address is associated with anyone of said plurality of secured computers, then the method further comprises the step of permitting said executable packet to proceed.

3. The method of claim 1 wherein if the determining step a) determines within said first degree of certainty or is uncertain whether said source address is associated with anyone of said plurality of secured computers and if the determining step b) determines within said second degree of certainty that said destination address is not associated with anyone of said plurality of secured computers, then the method further comprises the step of permitting said executable packet to proceed.

4. The method of claim 1 wherein if the determining step a) determines within said first degree of certainty or is uncertain whether said source address is not associated with anyone of said plurality of secured computers and if the determining step b) determines within said second degree of certainty or is uncertain whether said destination address is associated with anyone of said plurality of secured

computers, then the method further comprises the step of prohibiting said executable packet from proceeding.

5. The method of claim 1 wherein said executable packet includes an applet.

6. The method of claim 1 wherein said determining steps a) and b) are executed by an intelligent firewall associated with said plurality of secured computers.

7. A method for determining the trust worthiness of executable packets in a computer network having a plurality of secured computers and a plurality of unsecured computers, each executable packet having a source address and a destination address, said method comprising the step of:

determining within a degree of certainty whether a source address of one said executable packet is associated with anyone of said plurality of secured computers, said source address is not associated with anyone of said plurality of secured computers, or association of said source address with anyone of said plurality of secured computers is uncertain.

8. The method of claim 7 wherein if the determining step determines within said degree of certainty that said source address is associated with anyone of said plurality of secured computers, then the method further comprises the step of permitting said executable packet to proceed.

9. The method of claim 7 wherein said executable packet includes an applet.

10. The method of claim 7 wherein said determining step is executed by an intelligent firewall associated with said plurality of secured computers.

11. A method for determining the trust worthiness of executable packets in a computer network having a plurality of secured computers and a plurality of unsecured computers, each executable packet having a source address and a destination address, said method comprising the step of:

determining within a degree of certainty whether a destination address of one said executable packet is associated with anyone of said plurality of secured computers, said destination address is not associated with anyone of said plurality of secured computers, or association of said destination address with anyone of said plurality of secured computers is uncertain.

12. The method of claim 11 wherein if the determining step determines within said degree of certainty that said destination address is not associated with anyone of said plurality of secured computers, then the method further comprises the step of permitting said executable packet to proceed.

13. The method of claim 11 wherein said executable packet includes an applet.

14. The method of claim 11 wherein said determining step is executed by an intelligent firewall associated with said plurality of secured computers.

15. An intelligent firewall useful in association with a computer network having a plurality of secured computers and a plurality of unsecured computers, the firewall comprising:

a source address verifier configured to determine within a first degree of certainty whether a source address of an executable packet is associated with anyone of said plurality of secured computers, said source address is not associated with anyone of said plurality of secured computers, or association of said source address with anyone of said plurality of secured computers is uncertain.

16. The intelligent firewall of claim 15 further comprising:

- a destination address verifier configured to determine within a second degree of certainty whether a destination address of said executable packet is associated with anyone of said plurality of secured computers.

17. An intelligent firewall useful in association with a computer network having a plurality of secured computers and a plurality of unsecured computers, the firewall comprising:

- a destination address verifier configured to determine within a degree of certainty whether a destination address of an executable packet is associated with anyone of said plurality of secured computers, said destination address is not associated with anyone of said plurality of secured computers, or association of said destination address with anyone of said plurality of secured computers is uncertain.

18. A computer program product including a computer-usable medium having computer-readable code embodied therein configured to verify addresses of a plurality of executable packets for a computer network, the computer network including a plurality of secured computers and a plurality of unsecured computers, the computer-readable code comprising

- a computer-readable source address verifier configured to determine within a first degree of certainty whether a source address of one said executable packet is associated with anyone of said plurality of secured computers, said source address is not associated with anyone of said plurality of secured computers, or association of said source address with anyone of said plurality of secured computers is uncertain.

19. The computer program product of claim 18 wherein said computer-readable code further comprising:

- a computer-readable destination address verifier configured to determine within a second degree of certainty whether a destination address of said one executable packet is associated with anyone of said plurality of secured computers.

20. A computer program product including a computer-usable medium having computer-readable code embodied therein configured to verify addresses of a plurality of executable packets for a computer network, the computer network including a plurality of secured computers and a plurality of unsecured computers, the computer-readable code comprising:

- a computer-readable destination address verifier configured to determine within a degree of certainty whether a destination address of one said executable packet is associated with anyone of said plurality of secured computers, said destination address is not associated with anyone of said plurality of secured computers, or association of said destination address with anyone of said plurality of secured computers is uncertain.

* * * * *

EXHIBIT 7

US005623600A

United States Patent [19]

Ji et al.

[11] **Patent Number: 5,623,600**

[45] **Date of Patent: Apr. 22, 1997**

[54] **VIRUS DETECTION AND REMOVAL APPARATUS FOR COMPUTER NETWORKS**

[75] Inventors: Shuang Ji, Foster City; Eva Chen, Cupertino, both of Calif.

[73] Assignee: Trend Micro, Incorporated, Cupertino, Calif.

[21] Appl. No.: 533,706

[22] Filed: Sep. 26, 1995

[51] Int. Cl.^{6} G06F 11/34

[52] U.S. Cl. 395/187.01; 364/286.4; 364/DIG. 1

[58] Field of Search 395/186, 187.1, 395/200.06; 380/4; 364/285.1, 286.4

[56] References Cited

U.S. PATENT DOCUMENTS

4,975,950	12/1990	Lentz	380/4
5,319,776	6/1994	Hile et al.	395/187.01
5,414,833	5/1995	Hershey et al.	395/575
5,440,723	8/1995	Arnold et al.	395/181
5,444,850	8/1995	Chang	395/200
5,448,668	9/1995	Perelson et al.	395/182
5,452,442	9/1995	Kephart	395/183
5,485,575	1/1996	Chess et al.	395/183
5,491,791	2/1996	Glowny et al.	395/183
5,511,163	4/1996	Lerche et al.	395/183.15

FOREIGN PATENT DOCUMENTS

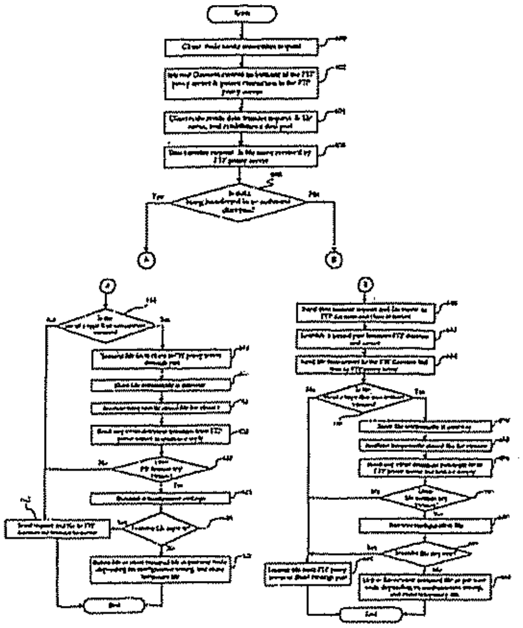
666671	8/1995	European Pat. Off.	H04L 29/06
6350784	6/1994	Japan	H04N 1/00
9322723	11/1993	WIPO	G06F 11/00

Primary Examiner—Robert W. Beausoliel, Jr.
Assistant Examiner—Albert Decady
Attorney, Agent, or Firm—Christopher M. Tobin; Greg T. Sueoka

[57] ABSTRACT

A system for detecting and eliminating viruses on a computer network includes a File Transfer Protocol (FTP) proxy server, for controlling the transfer of files and a Simple Mail Transfer Protocol (SMTP) proxy server for controlling the transfer of mail messages through the system. The FTP proxy server and SMTP proxy server run concurrently with the normal operation of the system and operate in a manner such that viruses transmitted to or from the network in files and messages are detected before transfer into or from the system. The FTP proxy server and SMTP proxy server scan all incoming and outgoing files and messages, respectively before transfer for viruses and then transfer the files and messages, only if they do not contain any viruses. A method for processing a file before transmission into or from the network includes the steps of: receiving the data transfer command and file name; transferring the file to a system node; performing virus detection on the file; determining whether the file contains any viruses; transferring the file from the system to a recipient node if the file does not contain a virus; and deleting the file if the file contains a virus.

22 Claims, 12 Drawing Sheets



Defendant's Trial Ex.
DTX - 1019
Case No. 06-369 GMS

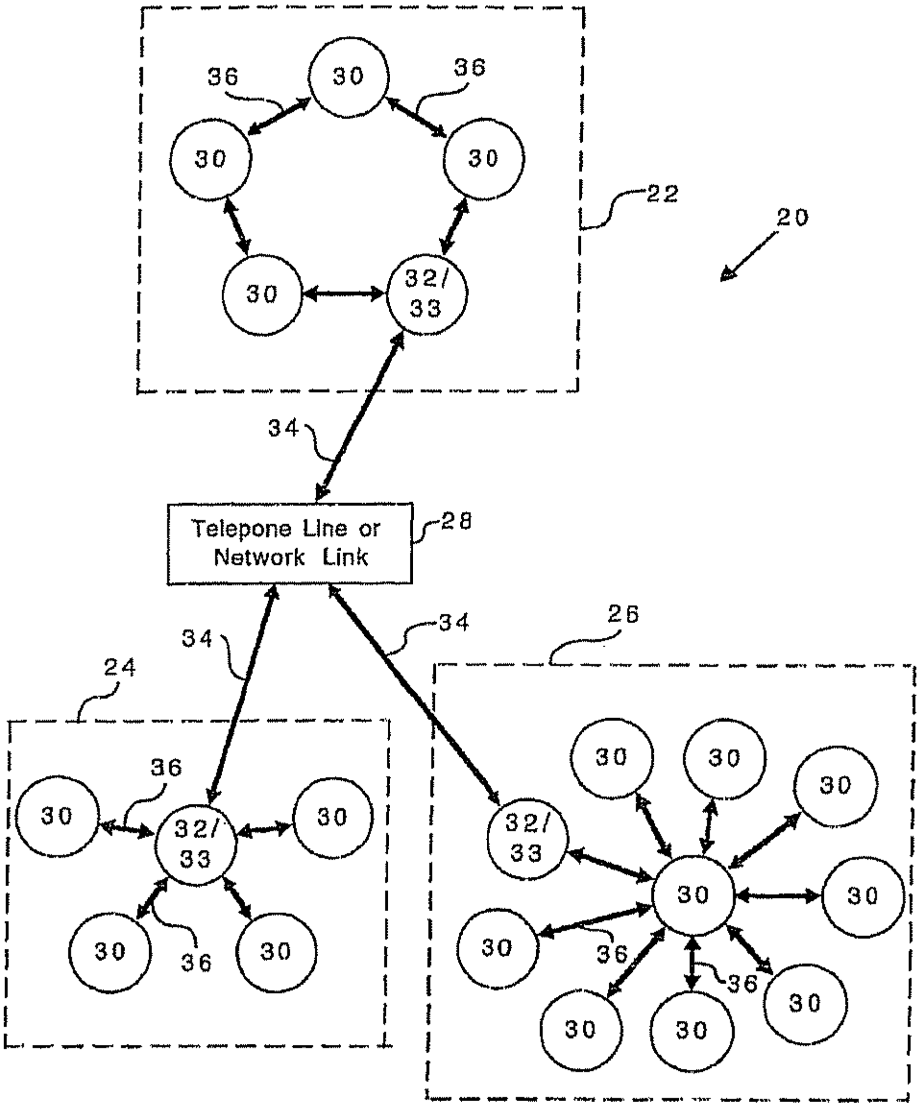


Fig. 1 (Prior Art)

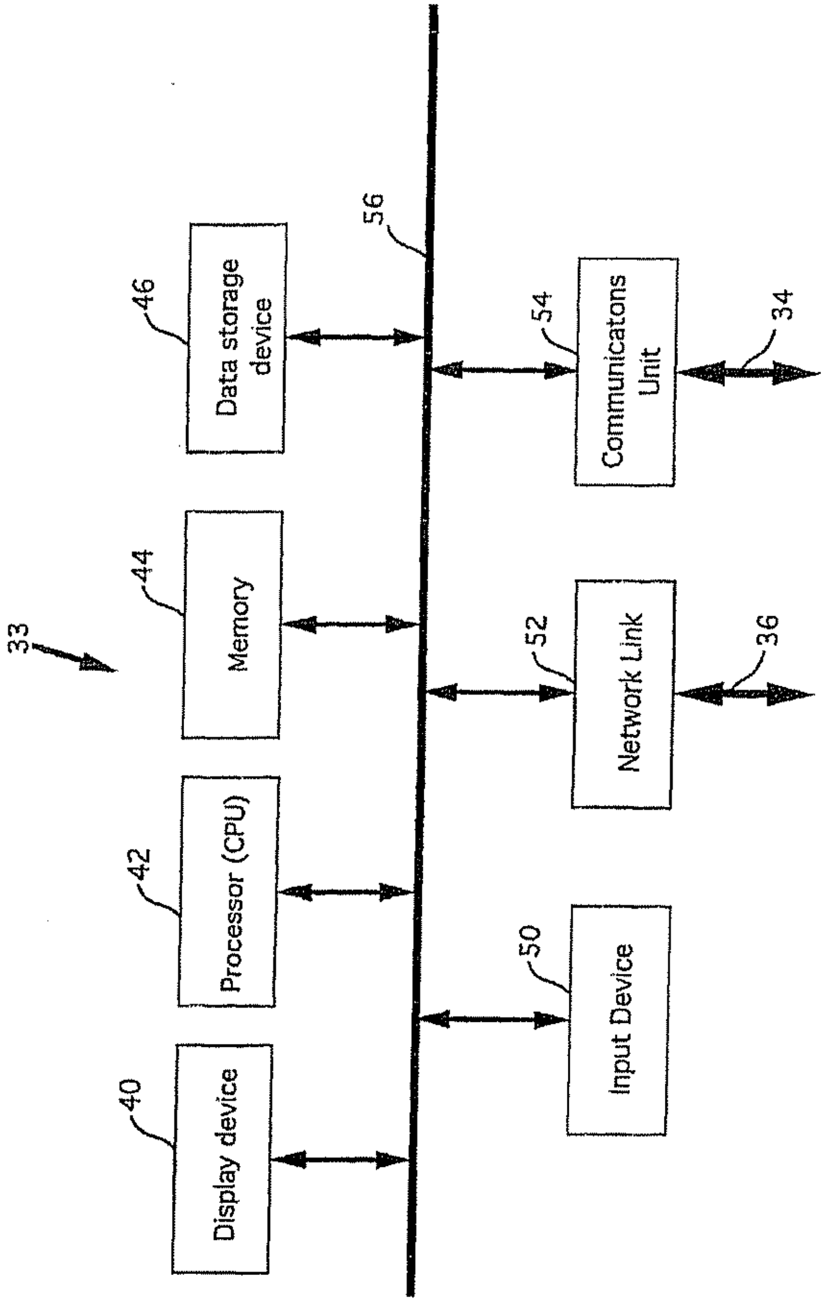

33
40
Display device
42
Processor (CPU)
44
Memory
46
Data storage device
56
50
Input Device
52
Network Link
36
54
Communicatons Unit
34

Fig. 2

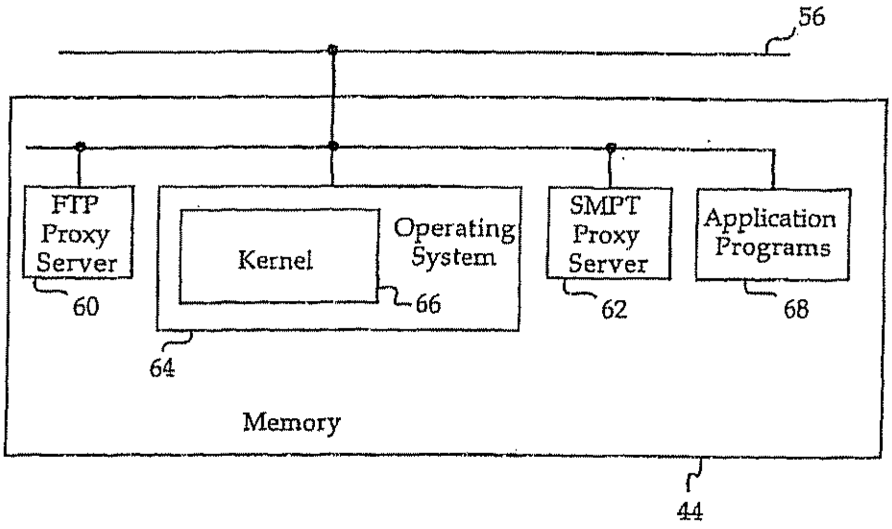

56
FTP Proxy Server
60
Kernel
Operating System
66
64
SMPT Proxy Server
62
Application Programs
68
Memory
44

FIG. 3

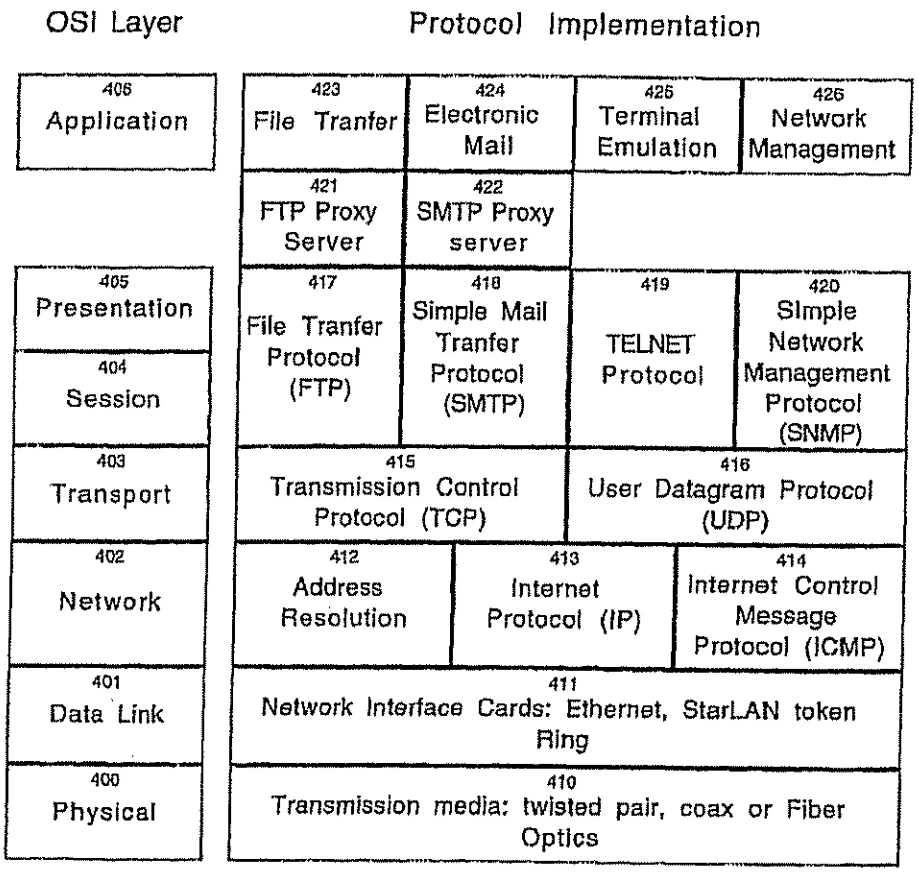

OSI Layer
Protocol Implementation
406 Application
423 File Tranfer
424 Electronic Mail
425 Terminal Emulation
426 Network Management
421 FTP Proxy Server
422 SMTP Proxy server
405 Presentation
404 Session
417 File Tranfer Protocol (FTP)
418 Simple Mail Tranfer Protocol (SMTP)
419 TELNET Protocol
420 Simple Network Management Protocol (SNMP)
403 Transport
415 Transmission Control Protocol (TCP)
416 User Datagram Protocol (UDP)
402 Network
412 Address Resolution
413 Internet Protocol (IP)
414 Internet Control Message Protocol (ICMP)
401 Data Link
411 Network Interface Cards: Ethernet, StarLAN token Ring
400 Physical
410 Transmission media: twisted pair, coax or Fiber Optics

FIG. 4

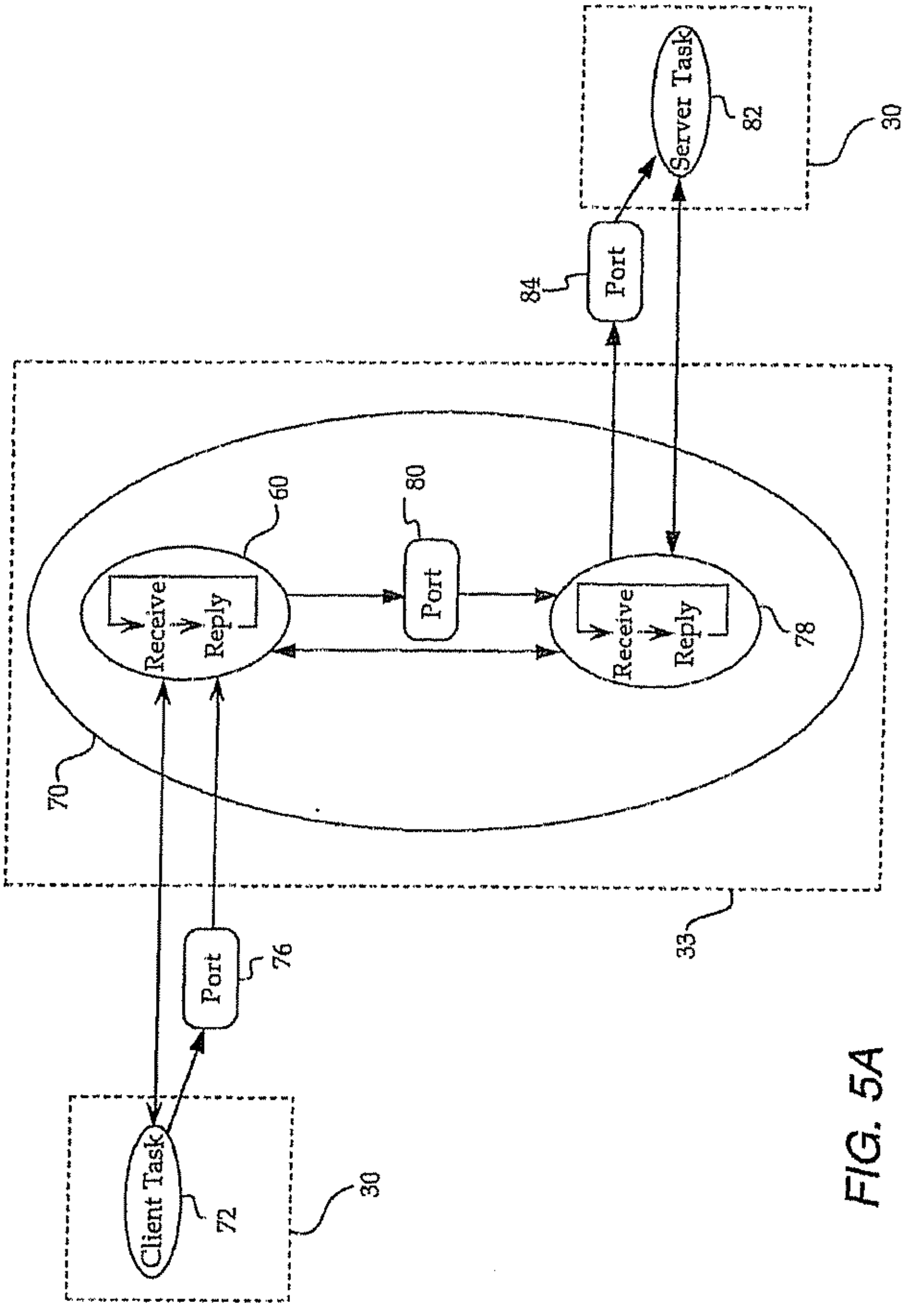

Server Task
82
30
Port
84
Receive
Reply
60
80
Port
Receive
Reply
78
70
33
Port
76
Client Task
72
30

FIG. 5A

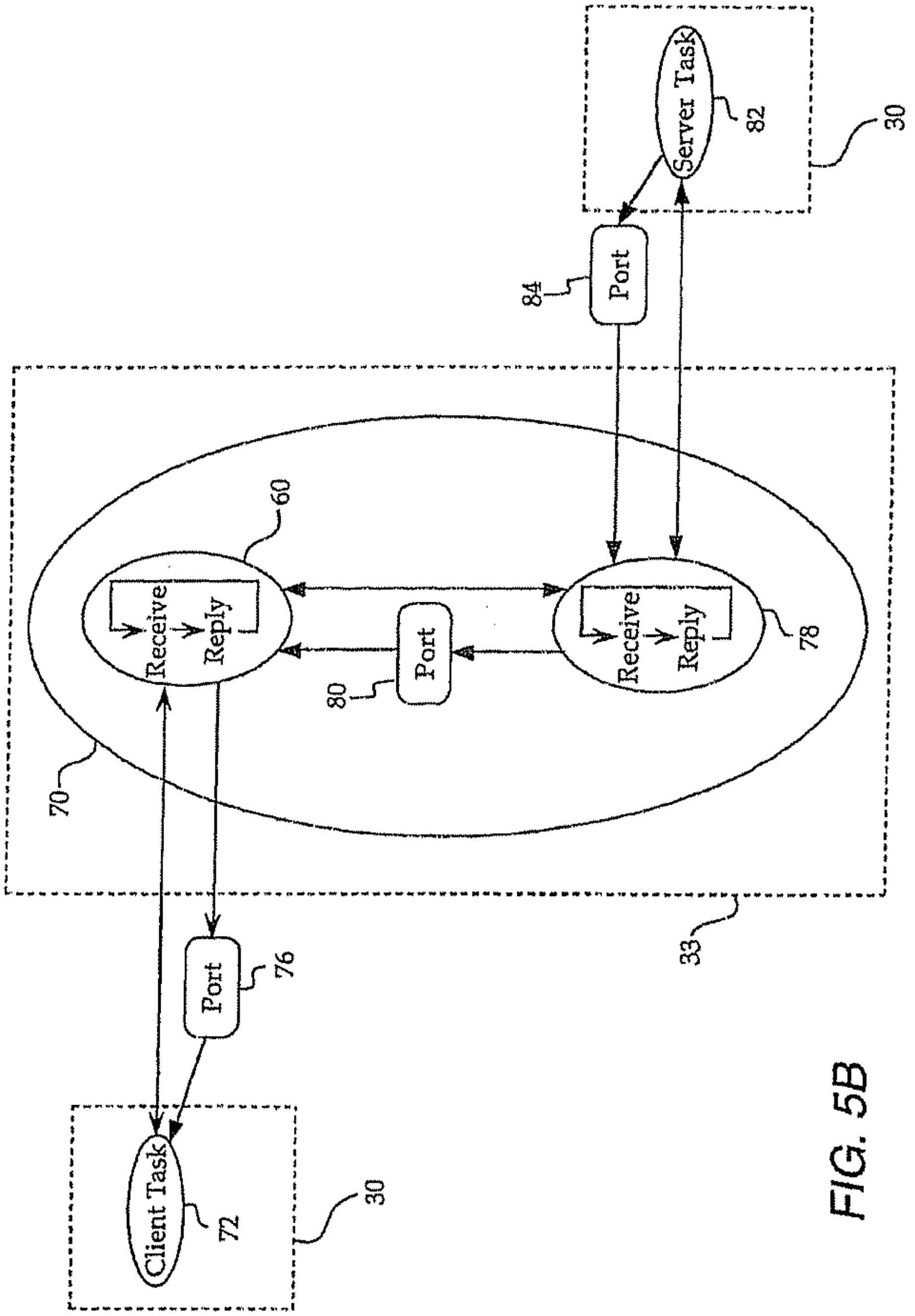

Server Task
82
30
Port
84
60
Receive
Reply
78
Port
80
70
33
Port
76
Client Task
72
30

FIG. 5B

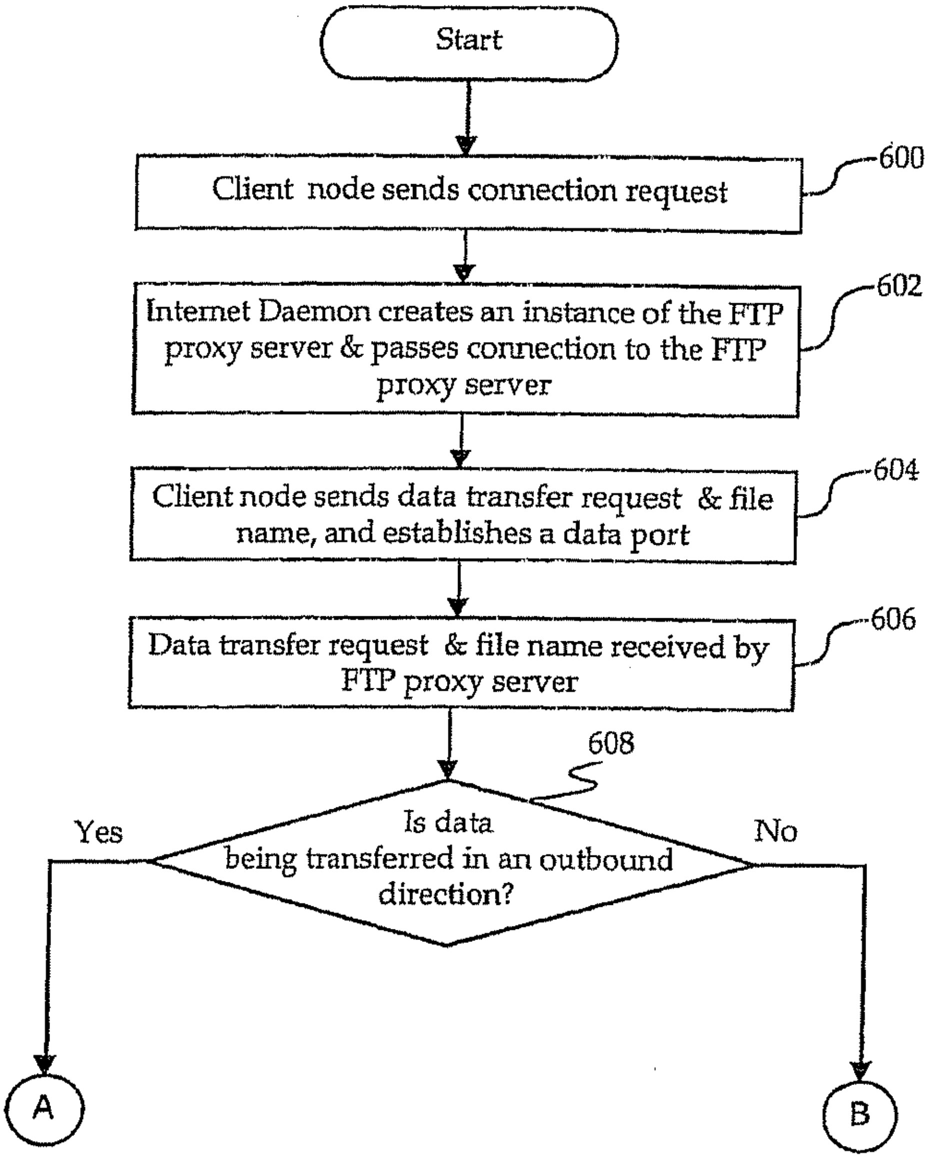

Start
Client node sends connection request
600
Internet Daemon creates an instance of the FTP proxy server & passes connection to the FTP proxy server
602
Client node sends data transfer request & file name, and establishes a data port
604
Data transfer request & file name received by FTP proxy server
606
608
Is data being transferred in an outbound direction?
Yes
No
A
B

FIG. 6A

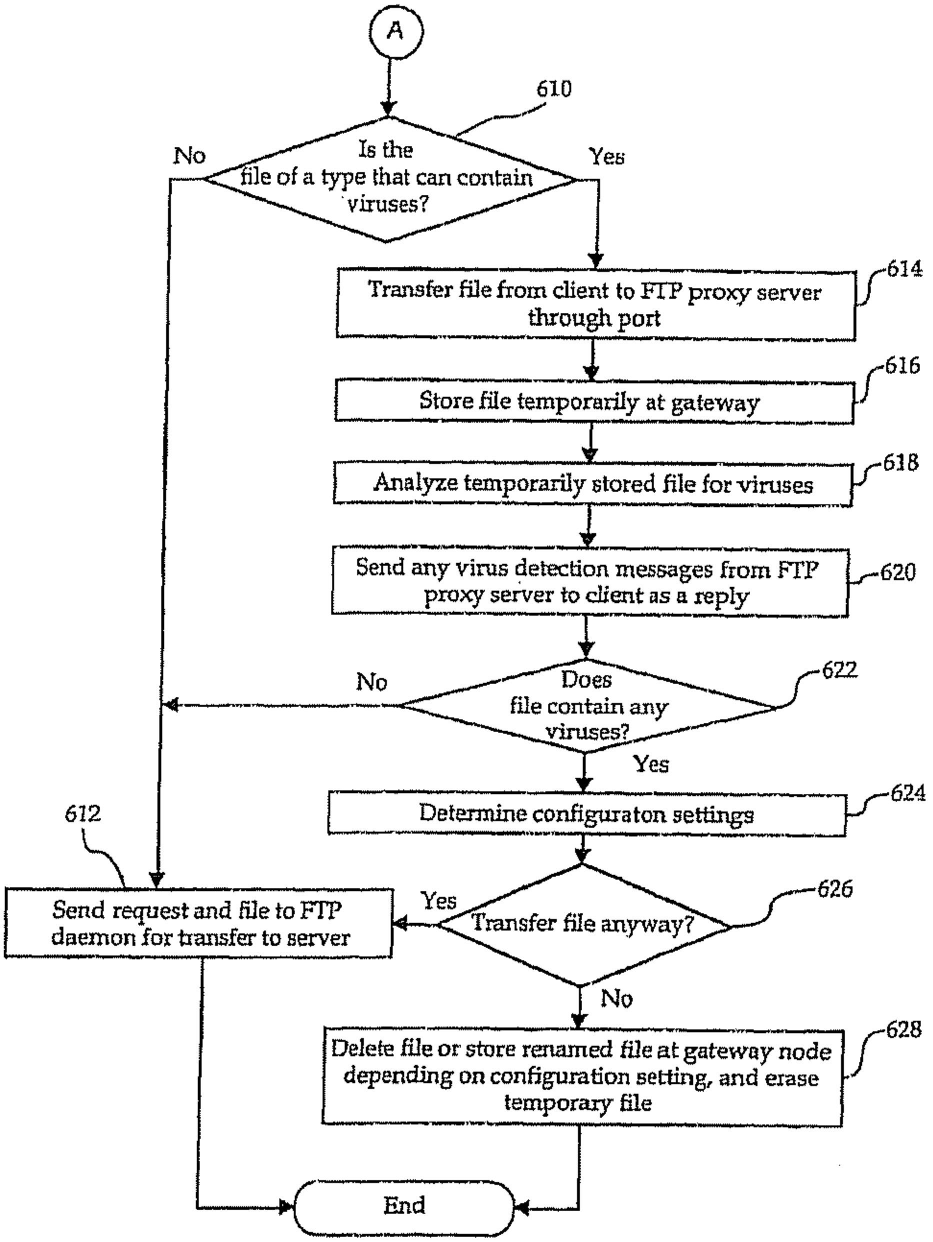

A
610
Is the file of a type that can contain viruses?
No
Yes
Transfer file from client to FTP proxy server through port
614
Store file temporarily at gateway
616
Analyze temporarily stored file for viruses
618
Send any virus detection messages from FTP proxy server to client as a reply
620
Does file contain any viruses?
622
No
Yes
Determine configuraton settings
624
612
Send request and file to FTP daemon for transfer to server
Yes
Transfer file anyway?
626
No
Delete file or store renamed file at gateway node depending on configuration setting, and erase temporary file
628
End

FIG. 6B

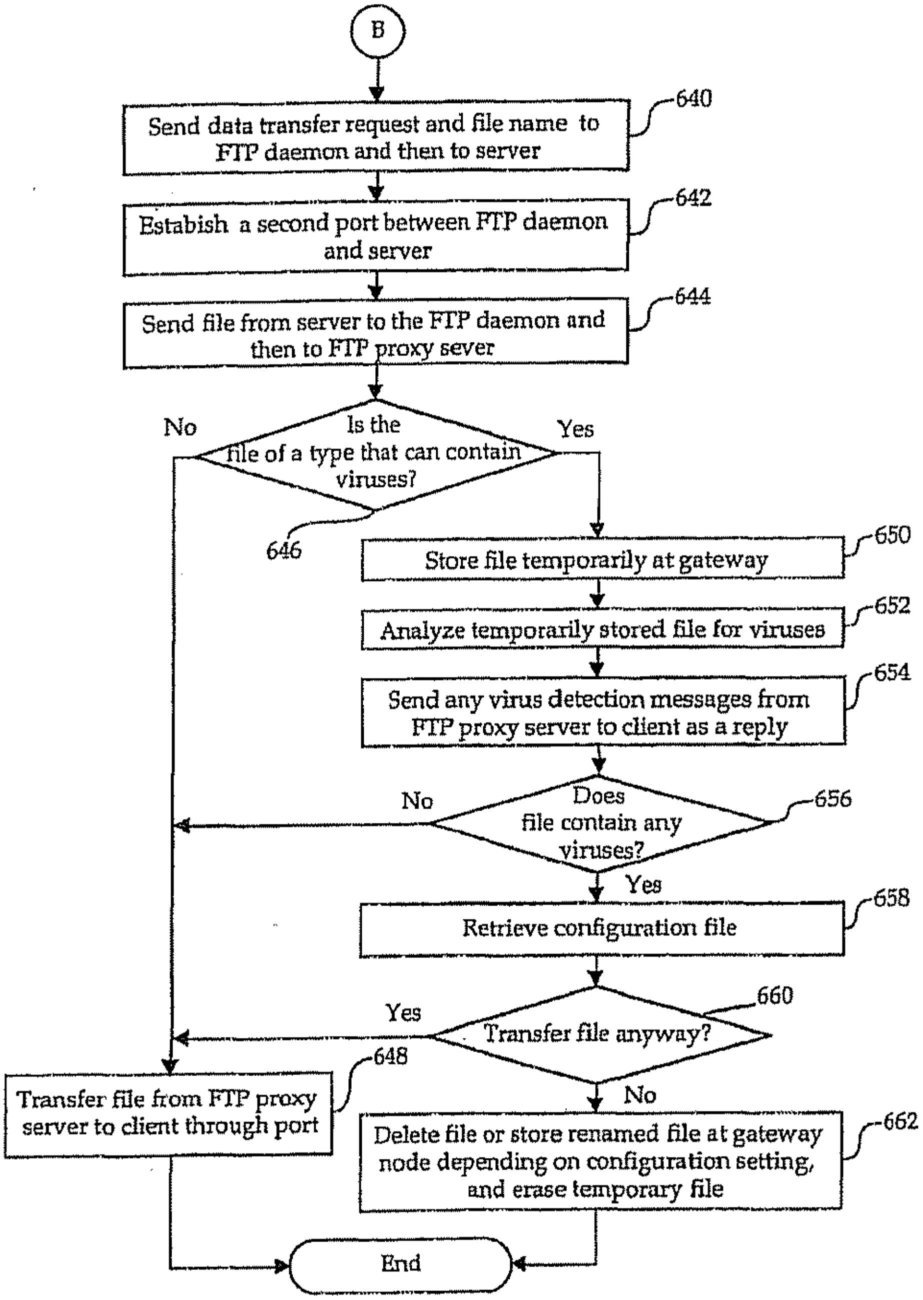

B
Send data transfer request and file name to FTP daemon and then to server
640
Estabish a second port between FTP daemon and server
642
Send file from server to the FTP daemon and then to FTP proxy sever
644
No
Is the file of a type that can contain viruses?
Yes
646
Store file temporarily at gateway
650
Analyze temporarily stored file for viruses
652
Send any virus detection messages from FTP proxy server to client as a reply
654
No
Does file contain any viruses?
656
Yes
Retrieve configuration file
658
Yes
Transfer file anyway?
660
648
Transfer file from FTP proxy server to client through port
No
Delete file or store renamed file at gateway node depending on configuration setting, and erase temporary file
662
End

FIG. 6C

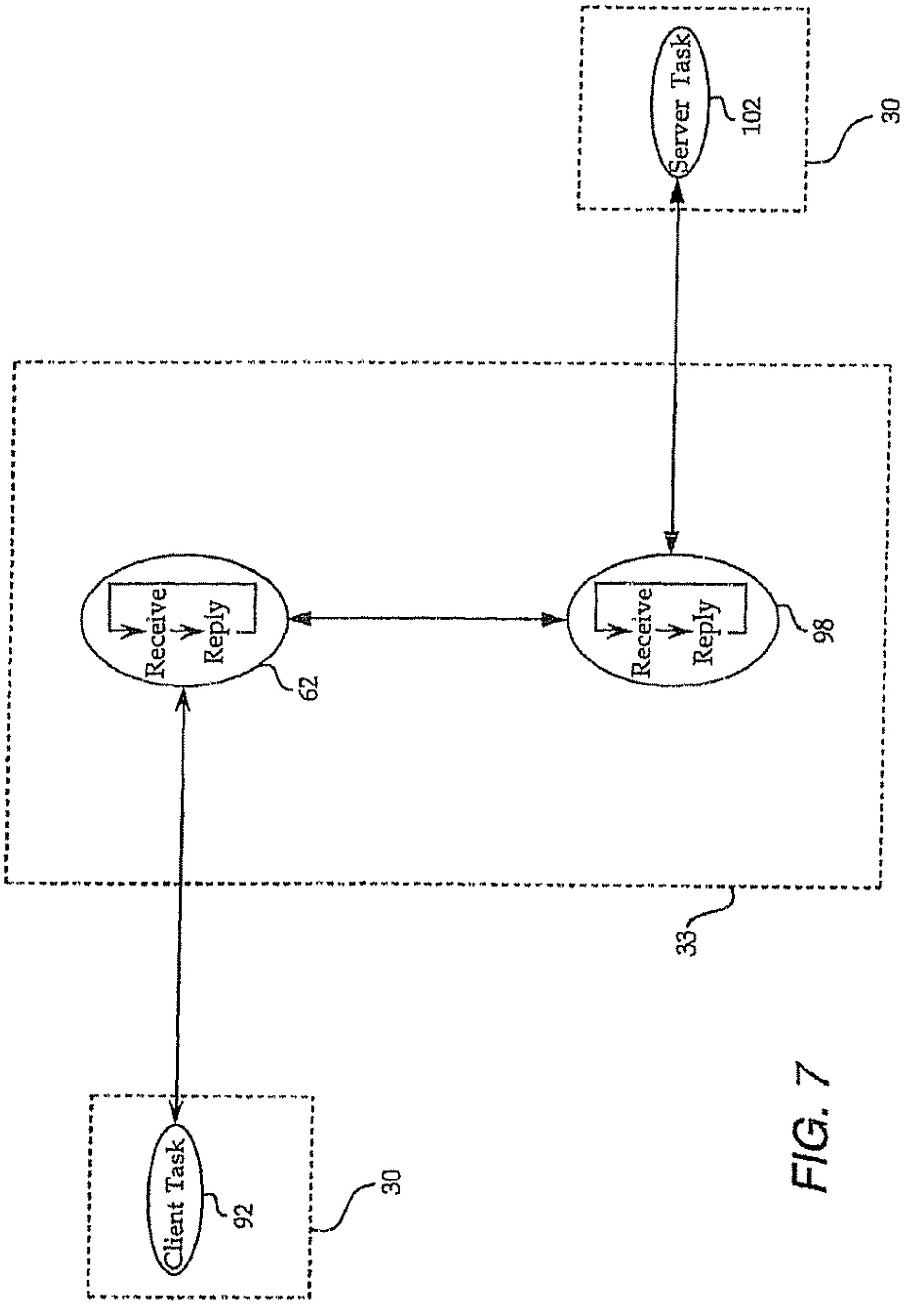

Server Task
102
30
Receive
Reply
98
Receive
Reply
62
33
Client Task
92
30

FIG. 7

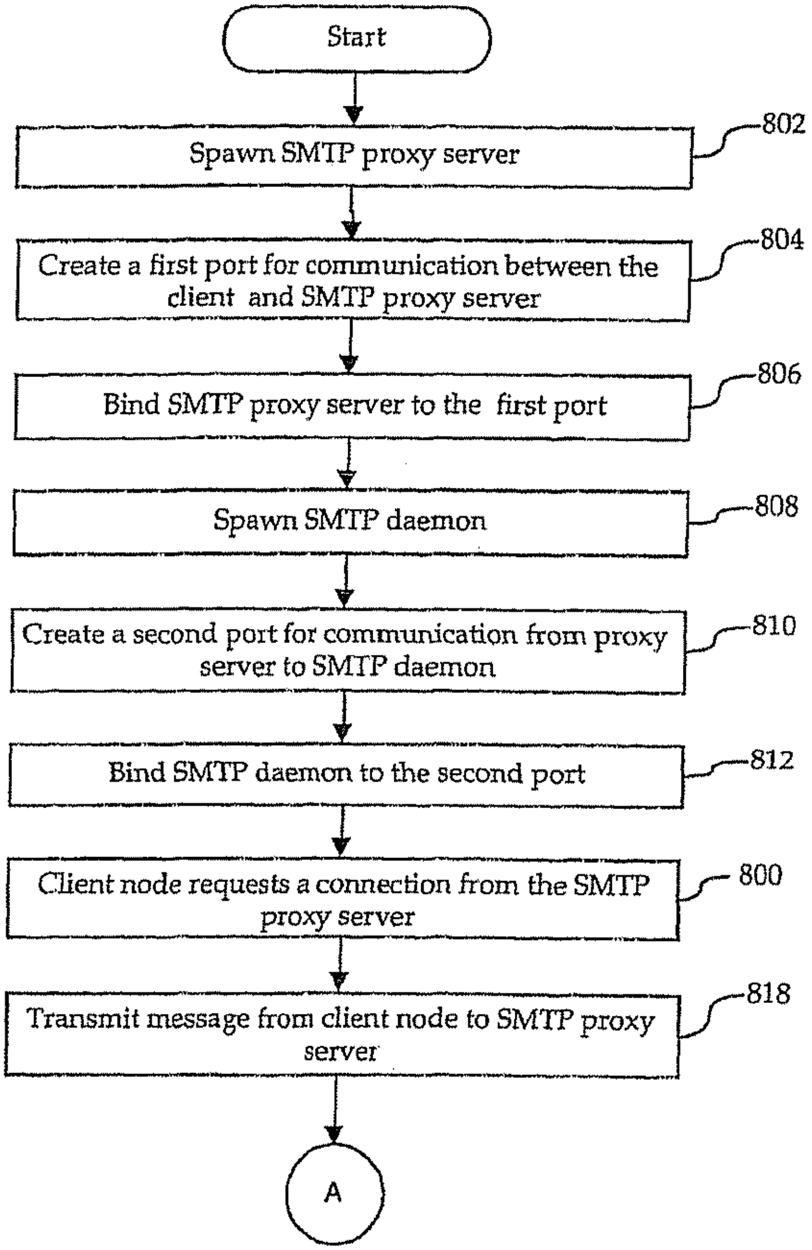

Start
Spawn SMTP proxy server
802
Create a first port for communication between the client and SMTP proxy server
804
Bind SMTP proxy server to the first port
806
Spawn SMTP daemon
808
Create a second port for communication from proxy server to SMTP daemon
810
Bind SMTP daemon to the second port
812
Client node requests a connection from the SMTP proxy server
800
Transmit message from client node to SMTP proxy server
818
A

FIG. 8A

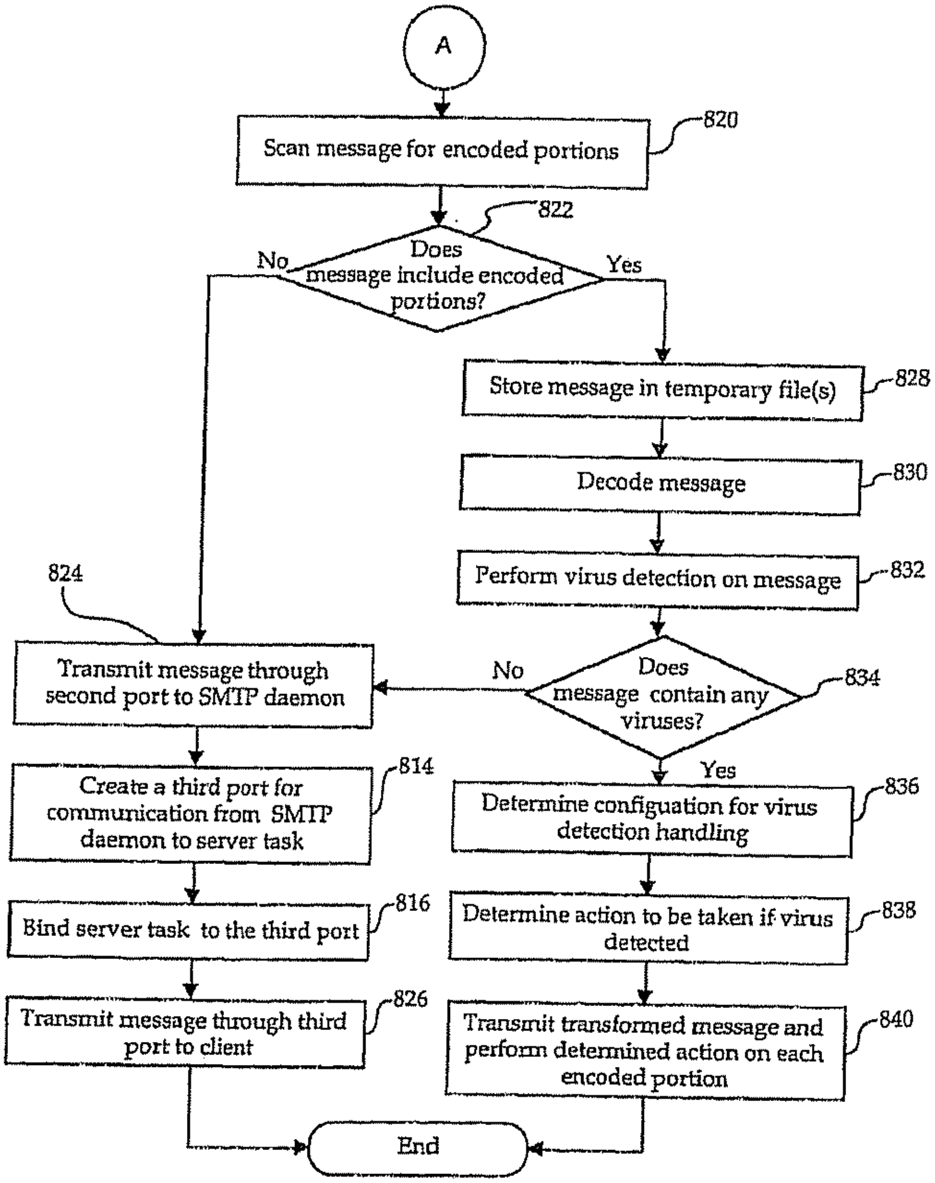

A
Scan message for encoded portions
820
822
Does message include encoded portions?
No
Yes
Store message in temporary file(s)
828
Decode message
830
Perform virus detection on message
832
824
Transmit message through second port to SMTP daemon
No
Does message contain any viruses?
834
Yes
Create a third port for communication from SMTP daemon to server task
814
Determine configuation for virus detection handling
836
Bind server task to the third port
816
Determine action to be taken if virus detected
838
Transmit message through third port to client
826
Transmit transformed message and perform determined action on each encoded portion
840
End

FIG. 8B

VIRUS DETECTION AND REMOVAL APPARATUS FOR COMPUTER NETWORKS

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates generally to computer systems and computer networks. In particular, the present invention relates to a system and method for detecting and removing computer viruses. Still more particularly, the present invention relates to a System and method for detecting and removing computer viruses from file and message transfers between computer networks.

2. Description of the Related Art

During the recent past, the use of computers has become widespread. Moreover, the interconnection of computers into networks has also become prevalent. Referring now to FIG. 1, a block diagram of a portion of a prior art information system 20 is shown. The portion of the information system 20 shown comprises a first network 22, a second network 24 and third network 26. This information system 20 is provided only by way of example, and those skilled in the art will realize that the information system 20 may include any number of networks, each of the networks being its own protected domain and having any number of nodes. As shown in FIG. 1, each of the networks 22, 24, 26 is formed from a plurality of nodes 30, 32. Each of the nodes 30, 32 is preferably a microcomputer. The nodes 30, 32 are coupled together to form a network by a plurality of network connections 36. For example, the nodes 30, 32 may be connected together using a token ring format, ethernet format or any of the various other formats known in the art. Each of the networks 22, 24, 26 includes a node 32 that acts as a gateway to link the respective network 22, 24, 26 to other networks 22, 24, 26. Each of the gateway nodes 32 is preferably coupled by a standard telephone line connection 34 such as POTS (Plain Old Telephone Service) or a T-1 link to the other gateway nodes 32 through a telephone switching network 28. All communication between the networks 22, 24, 26 is preferably performed through one of the gateway nodes 32.

One particular problem that has plagued computers, in particular microcomputers, have been computer viruses and worms. A computer virus is a section of code that is buried or hidden in another program. Once the program is executed, the code is activated and attaches itself to other programs in the system. Infected programs in turn copy the code to other programs. The effect of such viruses can be simple pranks that cause a message to be displayed on the screen or more serious effects such as the destruction of programs and data. Another problem in the prior art is worms. Worms are destructive programs that replicate themselves throughout disk and memory using up all available computer resources eventually causing the computer system to crash. Obviously, because of the destructive nature of worms and viruses, there is a need for eliminating them from computers and networks.

The prior art has attempted to reduce the effects of viruses and prevent their proliferation by using various virus detection programs. One such virus detection method, commonly referred to as behavior interception, monitors the computer or system for important operating system functions such as write, erase, format disk, etc. When such operations occur, the program prompts the user for input as to whether such an operation is expected. If such an operation is not expected (e.g., the user was not operating any program that employed such a function), the user can abort the operation knowing it was being prompted by a virus program. Another virus detection method, known as signature scanning, scans program code that is being copied unto the system. The system searches for known patterns of program code used for viruses. Currently, signature scanning only operates on the floppy disk drives, hard drives or optical drives. Yet another prior art approach to virus detection performs a checksum on all host programs stored on a system and known to be free from viruses. Thus, if a virus later attaches itself to a host program, the checksum value will be different and the presence of a virus can be detected.

Nonetheless, these approaches of the prior art suffer from a number of shortcomings. First, behavior interception is not successful at detecting all viruses because critical operations that may be part of the code for a virus can be placed at locations where such critical operations are likely to occur for the normal operation of programs. Second, most signature scanning is only performed on new inputs from disk drives. With the advent of the Internet and its increased popularity, there are no prior art methods that have been able to successfully scan connections 36 such as those utilized by a gateway node in communicating with other networks. Third, many of the above methods require a significant amount of computing resources, which in turn degrades the overall performance of system. Thus, operating the virus detection programs on every computer becomes impractical. Therefore, the operation of many such virus detection programs is disabled for improved performance of individual machines.

Therefore, there is a need for a system and method for effectively detecting and eliminating viruses without significantly effecting the performance of the computer. Moreover, there is a need for a system and method that can detect and eliminate viruses in networks attached to other information systems by way of gateways or the Internet.

SUMMARY OF THE INVENTION

The present invention overcomes the limitations and shortcomings of the prior art with an apparatus and method for detecting and eliminating viruses on a computer network. A system including the present invention is a network formed of a plurality of nodes and a gateway node for connection to other networks. The nodes are preferably microcomputers, and the gateway node comprises: a display device, a central processing unit, a memory forming the apparatus of the present invention, an input device, a network link and a communications unit. The memory further comprises an operating system including a kernel, a File Transfer Protocol (FTP) proxy server, and a Simple Mail Transfer Protocol (SMTP) proxy server. The central processing unit, display device, input device, and memory are coupled and operate to execute the application programs stored in the memory. The central processing unit of the gateway node also executes the FTP proxy server for transmitting and receiving files over the communications unit, and executes the SMTP proxy server for transmitting and receiving messages over the communications unit. The FTP proxy server and SMTP proxy server are preferably executed concurrently with the normal operation of the gateway node. The servers advantageously operate in a manner such that viruses transmitted to or from the network in messages and files are detected before the files are transferred into or from the network. The gateway node of the present invention is particularly advantageous because the impact of using the FTP proxy server and SMTP proxy server for the detection of viruses is minimized because only the files leaving or entering the network are evaluated for the presence of viruses and all other "intra" network traffic is unaffected.

The present invention also comprises a method for processing a file before transmission into the network and a method for processing a file before transmission from the network. The preferred method for processing a file comprises the steps of: receiving the data transfer command and file name; transferring the file to the proxy server; performing virus detection on the file; determining whether the file contains any viruses; transferring the file from the proxy server to a recipient node if the file does not contain a virus; and performing a preset action with the file if it does contain a virus. The present invention also includes methods for processing messages before transmission to or from the network that operate in a similar manner.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a prior art information system with a plurality of networks and a plurality of nodes upon which the present invention operates;

FIG. 2 is a block diagram of a preferred embodiment for a gateway node including tho apparatus of the present invention;

FIG. 3 is a block diagram of a preferred embodiment for a memory of the gateway node including the apparatus of the present invention;

FIG. 4 is a block diagram of a preferred embodiment for a protocol layer hierarchy constructed according to the present invention compared to the OSI layer model of the prior art;

FIG 5A is a functional block diagram showing a preferred system for sending data files according to a preferred embodiment of the present invention;

FIG. 5B is a functional block diagram showing a preferred system for receiving data files according to a preferred embodiment of the present invention;

FIGS. 6A, 6B and 6C are a flowchart of the preferred method for performing file transfer according to the present invention;

FIG. 7 is a functional block diagram showing a preferred system for transmitting mail messages according to a preferred embodiment of the present invention; and

FIGS. 8A and 8B are a flow chart of a preferred method for sending messages to/from a network.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

The virus detection system and method of the present invention preferably operates on an information system 20 as has been described above with reference to FIG. 1. The present invention, like the prior art, preferably includes a plurality of node systems 30 and at least one gateway node 33 for each network 22, 24, 26. However, the present invention is different from the prior art because it provides novel gateway node 33 that also performs virus detection for all files being transmitted into or out of a network. Furthermore, the novel gateway node 33 also performs virus detection on all messages being transmitted into or out of an associated network.

Referring now to FIG. 2, a block diagram of a preferred embodiment of the novel gateway node 33 constructed in accordance with the present invention is shown. A preferred embodiment of the gateway node 33 comprises a display device 40, a central processing unit (CPU) 42, a memory 44, a data storage device 46, an input device 50, a network link 52, and a communications unit 54. Tho CPU 42 is connected by a bus 56 to the display device 40, tho memory 44, the data storage device 46, the input device 50, the network link 52, and the communications unit 54 in a von Neumann architecture. The CPU 42, display device 40, input device 50, and memory 44 may be coupled in a conventional manner such as a personal computer. The CPU 42 is preferably a microprocessor such as an Motorola 68040 or Intel Pentium or X86 type processor; the display device 40 is preferably a video monitor; and the input device 50 is preferably a keyboard and mouse type controller. The CPU 42 is also coupled to the data storage device 44 such as a hard disk drive in a conventional manner. Those skilled in the art will realize that the gateway node 33 may also be a mini-computer or a mainframe computer.

The bus 56 is also coupled to the network link 52 to facilitate communication between the gateway node 33 and the other nodes 30 of the network. In the preferred embodiment of the present invention, the network link 52 is preferably a network adapter card including a transceiver that is coupled to a cable or line 36. For example, tho network link 52 may be an ethernet card connected to a coaxial line, a twisted pair line or a fiber optic line. Those skilled in the art will realize that a variety of different networking configurations and operating systems including token ring, ethernet, or arcnet may be used and that the present invention is independent of such use. The network link 52 is responsible for sending, receiving, and storing the signals sent over the network or within the protected domain of a given network. The network link 52 is coupled to the bus 56 to provide these signals to the CPU 34 and vice versa.

The bus 56 is also coupled to the communications unit 54 to facilitate communication between the gateway node 33 and the other networks. Specifically, the communications unit 54 is coupled to the CPU 42 for sending data and message to other networks. For example, tho communications unit 54 may be a modem, a bridge or a router coupled to the other networks in a conventional manner. In the preferred embodiment of the present invention, the communications unit 54 is preferably a router. The communications unit 54 is in turn coupled to other networks via a media 34 such as a dedicated T-1 phone line, fiber optics, or any one of a number of conventional connecting methods.

The CPU 42, under the guidance and control of instructions received from the memory 44 and from the user through the input device 50, provides signals for sending and receiving data using the communications unit 54. The transfer of data between networks is broken down into the sending and receiving files and messages which in turn are broken down into packets. The methods of the present invention employ a virus detection scheme that is applied to all transfers of messages and files into or out of a network via its gateway node 33.

Referring now to FIG. 3, the preferred embodiment of the memory 44 for the gateway node 33 is shown in more detail. The memory 44 is preferably a random access memory (RAM), but may also include read-only memory (ROM). The memory 44 preferably comprises a File Transfer Protocol (FTP) proxy server 60, a Simple Mail Transfer Protocol (SMTP) proxy server 62, and an operating system 64 including a kernel 66. The routines of the present invention for detecting viruses in file transfers and messages primarily include the FTP proxy server 60 and the SMTP proxy server 62. The FTP proxy server 60 is a routine for controlling file transfers to and from the gateway node 33 via the communications unit 54, and thus controlling file transfers to and from a given network of which the gateway node is a part. The operation of the FTP proxy server 60 is described below in more detail with reference to FIGS. 5A, 5B, 6A, 6B and 6C. Similarly, the SMTP proxy server 62 is a routine for controlling the transfer of messages to and from the gateway node 33, and thus to and from the respective network associated with the gateway node 33. The operation of the SMTP proxy server 62 is described below in more detail with reference to FIG. 7 8A and 8B. The present invention preferably uses a conventional operating system 28 such as Berkeley Software Distribution UNIX. Those skilled in the art will realize how the present invention may be readily adapted for use with other operating systems such as MACINTOSH System Software version 7.1, DOS, WINDOWS or WINDOWS NT. The memory 44 may also include a variety of different application programs 68 including but not limited to computer drawing programs, word processing programs, and spreadsheet programs. The present invention is particularly advantageous over the prior because it minimizes the impact of virus detection and elimination since the FTP proxy server 60 and SMTP proxy server 62 are preferably only included or installed in the memory 44 of the gateway nodes 33. Thus, all data being transferred inside the protected domain of a given network will not be checked because the data packets might not be routed via the gateway node 33.

While the apparatus of the present invention, in particular the FTP proxy server 60 and SMTP proxy server 62, has been described above as being located and preferably is located on the gateway node 33, those skilled in the art will realize that the apparatus of the present invention could also be included on a FTP server or a world wide web server for scanning files and messages as they are downloaded from the web. Furthermore, in an alternate embodiment, the apparatus of the present invention may be included in each node of a network for performing virus detection on all messages received or transmitted from that node.

As best shown in FIG. 4, the CPU 42 also utilizes a protocol layer hierarchy to communicate over the network. The protocol layers of the hierarchy of the present invention are shown in FIG. 4 in comparison to the ISO-OSI reference model, for example. The protocol layers 410–426 of the hierarchy of the present invention are similar to the prior art protocol layers for the lower four layers 400–403 including: (1) a physical layer 400 formed of the transmission media 410; (2) a data link layer 401 formed of the network interface cards 411; (3) a network layer 402 formed of address resolution 412, Internet protocol 413 and Internet control message protocol 414; and (4) a transport layer 403 formed of the transmission control protocol 415 and a user datagram protocol 416. Corresponding to the presentation 405 and session 404 layers, the protocol hierarchy of the present invention provides four methods of communication: a file transfer protocol 417, a simple mail transfer protocol 419, a TELNET protocol 419 and a simple network management protocol 420. There are corresponding components on the application layer 406 to handle file transfer 423, electronic mail 424, terminal emulation 425, and network management 426. The present invention advantageously detects, controls and eliminates viruses by providing an additional layer between the application layer 406 and the presentation layer 405 for the gateway nodes 33. In particular, according to the hierarchy of the present invention, a FTP proxy server layer 421 and a SMTP proxy server layer 422 are provided. These layers 421,422 operate in conjunction with the file transfer layer 423 and file transfer protocol 417, and the electronic mail layer 424 and the SMTP protocol layer 418, to process file transfers and messages, respectively. For example, any file transfer requests are generated by the file transfer application 423, first processed by the FTP proxy server layer 421, then processed by the file transfer protocol 417 and other lower layers 415, 413, 411 until the data transfer is actually applied to the transmission media 410. Similarly, any messaging requests are first processed by the SMTP proxy server layer 418, and thereafter processed by the SMTP protocol and other lower layers 415, 413, 411 until the physical layer is reached. The present invention is particularly advantageous because all virus screening is performed below the application level. Therefore, the applications are unaware that such virus detection and elimination is being performed, and these operations are completely transparent to the operation of the application level layers 406. While the FTP proxy server layer 421 and the SMTP proxy server layer 422 have been shown in FIG 4 as being their own layer to demonstrate the coupling effects they provide between the file transfer layer 423 and file transfer protocol 417, and the electronic mail layer 424 and the SMTP protocol layer 418, those skilled in the art will realize that the FTP proxy server layer 421 and the SMTP proxy server layer 422 can also be correctly viewed as being part of the file transfer protocol layer 417 and the SMTP protocol layer 418, respectively, because they are invisible or transparent to the application layer 406.

A preferred method of operation and an embodiment for the FTP proxy server 60 will be described focusing on its relationship to and its control of the gateway node 33, and thus, control over access to the medium, line 34, for connections to other networks. The method can best be understood with reference to FIGS. 5A and 5B, that graphically show the functions performed by an Internet daemon 70, the FTP proxy server 60, and an FTP daemon 78. each of which resides on the gateway note 33. In FIGS. 5A and 5B, like reference numbers have been used for like parts and the figures are different only in the direction in which the file is being transferred (either from client task 72 to server task 82 or from server task 82 to client task 72). For the sake of clarity and ease of understanding only the data ports are shown in FIGS. 5A and 5B, and the bi-directional lines represent command or control pathways and are assumed to include a command port although it is not explicitly shown. The operation FTP proxy server 60 will now be described with reference to a file transfer between a client task 72 (requesting machine) and a server task 82 (supplying machine). While it is assumed that the client task 72 (requesting machine) is inside a protected domain and the server task 82 (supplying machine) is outside the protected domain, the invention described below is also used by the gateway node 33 when client task 72 (requesting machine) is outside the protected domain and the server task 82 (supplying machine) is inside the protected domain.

FIGS. 6A–6C are a flowchart of a preferred method for performing file transfers from a controlled domain of a network across a medium 34 to another network (e.g., a file transfer from a node 32 of the second network 24 across the media 34 to a second node 32 of the third network 26). The method begins with step 600 with the client node sending a connection request over the network to the gateway node 33. In step 602, The gateway node 33 preferably has an operating system 64 as described above, and part of the operating system 64 includes a fire wall, or program including routines for authenticating users. The gateway node 33 first tries to authenticate the user and decide whether to allow the connections requested, once the request is received. This is done in a conventional manner typically available as part of UNIX. The Internet daemon 70 creates an instance of the FTP proxy server 60 and passes the connection to the FTP proxy server 60 for servicing in step 602. The Internet daemon 70 is program that is part of the operating system 64, and it runs in the background. When being run, one of the functions of the Internet daemon 70 is to bind socket ports for many well-known services, such as TELNET, login, and FTP. When a connect request is detected, the Internet daemon 70 constructed in accordance with the present invention, spawns the FTP proxy server 60, which is the server that will actually handle the data transfer. Thereafter, the FTP proxy server 60 controls the network traffic passing between the client task 72 and the server task 82. Then in step 604, the client node sends a data transfer request and file name, and established a first data port 76 through which the data will be transferred between the FTP proxy server 60 and the client task 72. In step 606 the data transfer request and file name are received by the FTP proxy server 60. In step 608, the FTP proxy server 60 determines whether the data is being transferred in an outbound direction (e.g., the file is being transferred from the client task 72 to the server task 82). This can be determined by the FTP proxy server 60 by comparing the data transfer request. For example, if the data transfer request is the STOR command then the data is being transferred in an outbound direction; and if the data transfer request is the RETR command then the data is not being transferred in an outbound direction.

If the data is being transferred in an outbound direction, then the method transitions from step 608 to step 610. Referring now to FIG. 6B in conjunction with FIG. 5A, the process for transferring data out of the protected domain of the network is described in more detail. In step 610, the FTP proxy server 60 determines whether the file to be transferred is of a type that can contain viruses. This step is preferably performed by checking the extension of the file name. For example, txt, .bmd. .pcx and .gif extension files indicate that the file is not likely to contain viruses while .exe, .zip, and .com extension files are of the type that often contain viruses. If the file to be transferred is not of a type that can contain viruses, then the method continues in step 612. In step 612, a second data port 80 is established and the data transfer request & the file are sent from the FTP proxy server 60 to the FTP daemon 78 so that the file can be sent to the server task 82. The FTP daemon 78 is a program executed by the gateway node 33 that communicates the transfer commands to the server task 82, establishes a third port 84 for sending the file including binding the server task 82 and FTP daemon 78 to the third port 84, and transmits the file to the server task 82. Once transmitted, the method is complete and ends. However, if it is determined in step 610 that the file to be transferred is of a type that can contain viruses, the method proceeds to step 614. In step 614, the FTP proxy server 60 transfers the file from the client to the FTP proxy server 60 through the first port 76, and in step 616, the file is temporarily stored at the gateway node 33. Then in step 618, the temporarily stored file is analyzed to determine if it contains viruses. This is preferably done by invoking a virus-checking program on the temporarily stored file. For example, a program the performs a version of signature scanning virus detection such as PC-Cillin manufactured and sold by Trend Micro Devices Incorporated of Cupertino, Calif. may be used. However, those skilled in the art will realize that various other virus detection methods may also be used in step 618. In step 620, output of the virus checking program is preferably echoed to the user/client task 72 by the FTP proxy server 60 as part of a reply message. Next in step

622, the method determines whether any viruses were detected. If no viruses are detected, the method continues in step 612 and transmits the file as has been described above. However, if a virus is detected, the present invention advantageously allows the FTP proxy server 60 to respond in any number of a variety of ways. The response of the FTP proxy server 60 is determined according to user's needs and wants as specified in a configuration file. This configuration file is preferably fully modifiable according to input from the user and stored in memory 44. For example, some options the user might specify are: 1) to do nothing and transfer the file; 2) to delete or erase the temporary file and do not transfer the file; or 3) to rename the file and store it in a specified directory on the gateway node 33 and notify the user of the new file name and directory path which can used to manually request the file from the system administrator. Those skilled in the art will realize that there are variety of other alternatives that users might specify, and steps 624, 626, and 628 are provided only by way of example. Next in step 624, the configuration file is retrieved to determine the handling of the temporary file. In step 626, the FTP proxy server 60 determines if it is to ignore the existence of a virus and a continue the transfer. If so, the method continues in step 612 where the file is passed to the FTP daemon 78 and the temporary file is deleted. If not the method continues to step 628 where either the file is deleted and not sent to the server task 82, and the temporary file is erased from the gateway node 33; or the file is renamed and stored in a specified directory on the gateway node 33 and the user is notified of the new file name and directory path which can used to manually request the file from the system administrator, and the temporary file is erased the gateway node 33. The action taken in step 628 depends on the configuration settings as determined in step 624. After step 628, the method ends. As can be seen from FIG. 5A, the path for the file is from client task 72 through the first data port 76 to the FTP proxy server 60, then to the FTP daemon 78 through the second data port 80 and finally to the server task 82 through the third data port 84.

Referring back to step 608 of FIG. 6A, if the data is not being transferred in an outbound direction, then the method transitions from step 608 to step 640. Referring now to FIG. 6C in conjunction with FIG. 5B, the process for transferring data into the protected domain of the network is described in more detail. In step 640, the FTP proxy server 60 next sends the data transfer request and file name first to the FTP daemon 78 and then on to the server task 82. In step 642, a second port 80 is established between the FTP proxy server 60 and the FTP daemon 78. Then a third data port 84 is established between the FTP daemon 78 and the server task 82. Both ports 80, 84 are established similar to the establishment of the first port 76. The FTP daemon 78 will request and obtain the third port 84 from the Internet daemon 70, and send a port command to the server task 82 including an address for the third port 84. The server task 82 will then connect to the third port 84 and begin the data transfer in step 644. The FTP daemon 78 in turn sends the file to the FTP proxy server 60. Next in step 646, the FTP proxy server 60 determines whether the file to be transferred is of a type that can contain viruses. This is done the same was as described above with reference to step 610. If the file to be transferred is not of a type that can contain viruses, then the method continues in step 648 where the file is transferred from the FTP proxy server 60 through the first port 76 to the client task 72, then the method is complete and ends. On the other hand, if the file to be transferred is a type that can contain viruses, the method in step 650 temporarily stores the file at the gateway node. Then in step 652, the temporarily stored file is analyzed to determine if it contains viruses. The analysis here is the same as step 618. In step 652, the output of the virus checking program is preferably echoed to the client task 72 by the FTP proxy server 60 as part of a reply message. Next in step 656, the method determines whether any viruses were detected. If no viruses are detected, the method continues in step 648 as has been described above. However, if a virus is detected, the present invention retrieves the configuration file to determine the handling of the temporary file. In step 660, the FTP proxy server 60 determines if it is to ignore the existence of a virus and a continue the. file transfer. If so the method continues in step 648 where the file is passed to the client task 72 and the temporary file is erased. If not the method continues to step 662 where the temporary file is erased, and the file is either deleted and not sent to the client task 72 or the file is renamed, stored on the gateway node 33, and the client task 72 is notified of new name and path so that the file may be manually retrieved by the system administrator. The method then ends. As can be seen from FIG. 5B, the data transfer request is passed from the client task 72, to the FTP proxy server 60, then to the FTP daemon 78, and to the server task 82 which in response sends the file through the third port to the FTP daemon 78, and through the second port 80 on to the FTP proxy server 60, and finally through the first port 76 to the client task 72.

Referring now to FIGS. 7, 8A and 8B, the operation of the SMTP proxy server 62 will now be described. The SMTP proxy server 62 controls the only other entry channel through which data, and therefore viruses, can enter the protected domain of a given network. The SMTP proxy server 62 is preferably a program that resides on the gateway node 33, and controls and handles all transfers of electronic messages or mail in and out of the network through the communications unit 54 and media' 34. While the SMTP proxy server 62 will now be described with reference to the transfer of a mail message from a client task 92 within the protected domain of the network to a server task 102 at a node on a different network outside the protected domain, those skilled in the art will understand how the SMTP proxy server 62 handles incoming mail messages in the same way. All mail messages are handled by the SMTP proxy server 62 in the same way and only the designation of which node 32 is the server and which is the client change depending on the direction the message is being sent from the perspective of the gateway node 33. Since mail messages are passed using the command pathways between nodes, only these pathways are shown in FIG. 7. For ease of understanding, the command ports have not been shown in FIG. 7, but will be discussed below in the relevant steps of the preferred method.

Referring now to FIG. 8A, the preferred method of the present invention for sending electronic mail begins in step 802 with the spawning or running the SMTP proxy server 62. Next in step 804, a first command port 96 for communication between the client task(s) 92 and the SMTP proxy server 62 is created. The address of the first port 96 along with a port command is provided to the SMTP proxy server 62. Then in step 806, the SMTP proxy server 62 is bound to the first port 96 to establish a channel for sending a mail message between any client tasks and the SMTP proxy server 62. Next in step 808, the SMTP proxy server 62 spawns a SMTP daemon 98 or SMTP server. The SMTP daemon 98 is preferably the existing program "sendmail" that is part of the BSD UNIX operating system. This is particularly advantageous because it reduces the amount of

code that needs to be written and assures compatibility with the lower layers of the OSI reference model. Then in step 810 a second command port is created for communication between the SMTP proxy server 62 and the SMTP daemon 98. In step 812, the SMTP daemon 98 is bound to the second command port for communication with the SMTP proxy server 62. Actually, the present invention binds the SMTP daemon 98 to the appropriate port, namely the second port by redefining the bind function in a shared library that is part of the operating system 64. The present invention advantageously exploits the fact that the SMTP daemon 98 (sendmail programs on most UNIX systems) are dynamically linked. The present invention utilizes a shared library which redefines the system call bind() and forces sendmail to link with the redefined version of the bind() call when executed. If the redefined version of the bind() call determines the SMTP daemon 98 (sendmail program) is trying to bind to the first command port (the smtp port), it will return to it a socket whose other end is the SMTP proxy server 62 (a socket to the second command port). Next in step 800, the client task 92 request a connection from the SMTP proxy server and is directed to used the first command port for communication. Then in step 818, the message is transmitted from the client task 92 through the first command port to the SMTP proxy server 62.

Referring now to FIG. 8B, the method continues in step 820 with the SMTP proxy server 62 scanning the message body and checking for any portions that are encoded. The present invention preferably scans the message for portions that have been encoded with an "uuencoded" encoding scheme that encodes binary data to ASCII data. "Uuencoded" portions of messages usually start with a line like "begin 644 filename," and end with a line like "end." The existence of such encoded portions suggests the possibility that a file may contain viruses. This scanning for "uuencoded" portions is just one of many scanning techniques that may be used, and those skilled in the art will realize that the present invention could be modified to scan for other encoded portions such as those encoded according to other schemes such as mime. Next in step 822, the SMTP proxy server 62 determines whether the message includes any encoded portions. If the message does not include any encoded portions, the SMTP proxy server 62 transmits the message through the second command port to the SMTP daemon 98 in step 824. Next in step 814, the SMTP daemon 98 creates a third command port for communication between the SMTP daemon 98 and the server task 102. Then in step 816 the server task 102 is bound to the third command port to establish communication between the server task 102 and the SMTP daemon 98. Those skilled in the art will realize that if the server task 102 resides on the gateway node 33, then steps 814 and 816 are not needed and may be omitted since no further transfer of data across the network is needed. Then the SMTP daemon 98 transmits the message through the third command port to the server task 102 in step 826 thereby completing the method.

On the other hand if in step 822 it is determined the message does include encoded portions, the SMTP proxy server 62 stores each of the encoded portions of the message in its own temporary file at the gateway node 33 in step 828. For example, if a message included three encoded portions, each encoded portion will be stored in a separate file. Then in step 830, each of the encoded portions stored in its own file is individually decoded using uudecode program, as will be understood by those skilled in the art. Such decoding programs known in the art convert the ASCII files back to their original binary code. Next in step 832, the SMTP proxy server 62 calls and executes a virus-checking program on each message portion stored in its temporary file(s). Then in step 834, the SMTP proxy server 62 determines whether any viruses were detected. If no viruses are detected, the method continues to steps 824, 814, 816 and 826 as has been described above. However, if a virus is detected, the present invention advantageously allows the SMTP proxy server 62 to respond in any number of a variety of ways, just as the FTP proxy server 60. The response of the SMTP proxy server 62 is also determined by the according to user's needs and wants as specified in a configuration file. This configuration file is preferably fully modifiable according to input from the user. The configuration for virus handling is determined in step 836. This could be done by retrieving and reading the configuration file or simply retrieving the configuration data already stored in memory 44. Then in step 838, the action to be taken is determined from the configuration settings. For example, some options the user might specify are: 1) to do nothing and transfer the mail message unchanged; 2) to transfer the mail message with the encoded portions that have been determined to have viruses deleted from the mail message; 3) rename the encode portions of the message containing viruses, store the renamed portions as files in a specified directory on the SMTP proxy server 62 and notify the user of the renamed files and directory path which can used to manually request the file from the system administrator; or 4) writing the output of step 832 into the mail message in place of the respective encoded portions and sending that mail message in steps 824 and 826. Once the action to be performed has been determined from examination of the configuration file, the specified action is taken in step 840, the transformed message is transmitted, the temporary file is erased, and the method ends. For example, if a message has three encoded portions, two encoded portions contain viruses, and the configuration file indicates that virus containing portions are to be deleted, then the method of the present invention would send a transformed message that was the same as the original message, but with the two encoded portions containing viruses deleted, to the server task 102.

While the present invention has been described with reference to certain preferred embodiments, those skilled in the art will recognize that various modifications may be provided. For example, the preferred operation of the present invention specifies that the FTP proxy server 60 determine whether the file type is one that can contain a virus (Steps 610 and 646). However, alternate embodiments can omit these steps and simply temporarily store and scan all files being transferred for viruses. Likewise the SMTP proxy server 60 may, in alternate embodiments, omit the step 822 of determining whether the message is encoded and temporarily store and scan all message being transmitted for viruses. Furthermore, while the invention has been described above as temporarily storing the file or message at the gateway node in a temporary file, this step could be omitted in the determination of whether a file includes a virus were done as the file was being transferred from the client node to the gateway node. These and other variations upon and modifications to the preferred embodiment are provided for by the present invention which is limited only by the following claims.

What is claimed is:

1. A system for detecting and selectively removing viruses in data transfers, the system comprising:

a memory for storing data and routines, the memory having inputs and outputs, the memory including a server for scanning data for a virus and specifying data handling actions dependent on an existence of the virus;

a communications unit for receiving and sending data in response to control signals, the communications unit having an input and an output;

a processing unit for receiving signals from the memory and the communications unit and for sending signals to the memory and communications unit; the processing unit having inputs and outputs; the inputs of the processing unit coupled to the outputs of memory and the output of the communications unit; the outputs of the processing unit coupled to the inputs of memory, the input of the communications unit, the processor controlling and processing data transmitted through the communications unit to detect viruses and selectively transfer data depending on the existence of viruses in the data being transmitted;

a proxy server for receiving data to be transferred, the proxy server scanning the data to be transferred for viruses and controlling transmission of the data to be transferred according to preset handing instructions and the presence of viruses, the proxy server having a data input a data output and a control output the data input coupled to receive the data to be transferred; and

a daemon for transferring data from the proxy server in response to control signals from the proxy server, the daemon having a control input, a data input and a data output the control input of the daemon coupled to the control output of the proxy server for receiving control signals, and the data input of the daemon coupled to the data output of the proxy server for receiving the data to be transferred.

2. The system of claim 1, wherein the proxy server is a FTP proxy server that handles evaluation and transfer of data files, and the daemon is an FTP daemon that communicates with a recipient node and transfers data files to the recipient node.

3. The system of claim 1, wherein the proxy server is a SMTP proxy server that handles evaluation and transfer of messages, and the daemon is an SMTP daemon that communicates with a recipient node and transfers messages to the recipient node.

4. A computer implemented method for detecting viruses in data transfers between a first computer and a second computer, the method comprising the steps of:

receiving at a server a data transfer request including a destination address;

electronically receiving data at the server;

determining whether the data contains a virus at the server;

performing a preset action on the data using the server if the data contains a virus;

sending the data to the destination address if the data does not contain a virus;

determining whether the data is of a type that is likely to contain a virus; and

transmitting the data from the server to the destination without performing the steps of determining whether the data contains a virus and performing a preset action if the data is not of a type that is likely to contain a virus.

5. The method of claim 4, further comprising the steps of storing the data in a temporary file at the server after the step of electronically transmitting; and wherein the step of determining includes scanning the data for a virus using the server.

6. The method of claim 5, wherein the step of scanning is performed using a signature scanning process.

7. The method of claim 4, wherein the step of performing a preset action on the data using the server comprises performing one step from the group of:

transmitting the data unchanged;

not transmitting the data; and

storing the data in a file with a new name and notifying a recipient of the data transfer request of the new file name.

8. The method of claim 4, wherein the step of determining whether the data is of a type that is likely to contain a virus is performed by comparing an extension type of a file name for the data to a group or known extension types.

9. The method of claim 4, further comprising the steps of:

determining whether the data is being transferred into a first network by comparing the destination address to valid addresses for the first network;

wherein the server is a FTP proxy server;

wherein the step of electronically receiving data comprises the steps of transferring the data from a client node to the FTP proxy server, if the data is not being transferred into the first network; and

wherein the step of electronically receiving data comprises the steps of transferring the data from a server task to an FTP daemon, and then from the FTP daemon to the FTP proxy server if the data is being transferred into the first network.

10. The method of claim 4, further comprising the steps of:

determining whether the data is being transferred into a first network by comparing the destination address to valid addresses for the first network;

wherein the server is a FTP proxy server;

wherein the step of sending the data to the destination address comprises transferring the data from the FTP proxy server to a node having the destination address, if the data is being transferred into the first network; and

wherein the step of sending the data to the destination address comprises transferring the data from the FTP proxy server to a FTP daemon, and then from an FTP daemon to a node having the destination address, if the data is not being transferred into the first network.

11. A computer implemented method for detecting viruses in a mail message transferred between a first computer and a second computer, the method comprising the steps of:

receiving a mail message request including a destination address;

electronically receiving the mail message at a server;

determining whether the mail message contains a virus, the determination of whether the mail message contains a virus comprising determining whether the mail message includes any encoded portions, storing each encoded portion of the mail message in a separate temporary file, decoding the encoded portions of the mail message to produced decoded portions of the mail message, scanning each of the decoded portions for a virus, and testing whether the scanning step found any viruses;

performing a preset action on the mail message if the mail message contains a virus; and

sending the mail message to the destination address if the mail message does not contains a virus.

12. The method of claim 11, wherein the step of determining whether the mail message includes any encoded portions searches for uuencoded portions.

13. A computer implemented method for detecting viruses in a mail message transferred between a first computer and a second computer, the method comprising the steps of:

receiving a mail message request including a destination address; electronically receiving the mail message at a server; scanning the mail message for encoded portions; determining whether the mail message contains a virus;

performing a preset action on the mail message if the mail message contains a virus;

sending the mail message to the destination address if the mail message does not contains a virus; and

wherein the step of sending the mail message to the destination address is performed if the mail message does not contain any encoded portions; the server includes a SMTP proxy server and a SMTP daemon; and the step of sending the mail message comprises transferring the mail message from the SMTP proxy server to the SMTP daemon and transferring the mail message from the SMTP daemon to a node having an address matching the destination address.

14. The method of claim 11, wherein the step of determining whether the mail message contains a virus, further comprises the steps of:

storing the message in a temporary file;

scanning the temporary file for viruses; and

testing whether the scanning step found a virus.

15. The method of claim 11, wherein step of scanning is performed using a signature scanning process.

16. The method of claim 11, wherein the step of performing a preset action on the mail message comprises performing one step from the group of:

transferring the mail message unchanged;

not transferring the mail message;

storing the mail message as a file with a new name and notifying a recipient of the mail message request of the new file name; and

creating a modified mail message by writing the output of the determining step into the modified mail message and transferring the mail message to the destination address.

17. The method of claim 11, wherein the step of performing a preset action on the mail message comprises performing one step from the group of:

transferring the mail message unchanged;

transferring the mail message with the encoded portions having a virus deleted; and

renaming the encode portions of the mail message containing a virus, and storing the renamed portions as files in a specified directory on the server and notifying a recipient of the renamed files and directory; and

writing the output of the determining step into the mail message in place of respective encoded portions that contain a virus to create a modified mail message and sending the modified mail message.

18. An apparatus for detecting viruses in data transfers between a first computer and a second computer, the apparatus comprising:

means for receiving a data transfer request including a destination address;

means for electronically receiving data at a server;

means for determining whether the data contains a virus at the server;

means for performing a preset action on the data using the server if the data contains a virus; and

means for sending the data to the destination address if the data does not contain a virus.

19 The apparatus of claim 18, wherein means for determining includes a means for scanning that scans the data using a signature scanning process.

20. The apparatus of claim 18, wherein the means for performing a preset action comprises:

means for transmitting the data unchanged;

means for not transmitting the data; and

means for storing the data in a file with a new name and notifying a recipient of the data transfer request of the new file name.

21. The apparatus of claim 18; further comprising:

a second means for determining whether the data is of a type that is likely to contain a virus; and

means for transmitting the data from the server to the destination without performing the steps of scanning, determining, performing and sending, if the data is not of a type that is likely to contain a virus.

22. The apparatus of claim 18, further comprising means for determining whether the data is being transferred into a first network by comparing the destination address to valid addresses for the first network.

* * * * *

EXHIBIT 8

US005951698A

United States Patent [19]

Chen et al.

[11] Patent Number: 5,951,698

[45] Date of Patent: Sep. 14, 1999

[54] SYSTEM, APPARATUS AND METHOD FOR THE DETECTION AND REMOVAL OF VIRUSES IN MACROS

[75] Inventors: Eva Y. Chen, Cupertino, Calif.; Jonny T. Ro, Taipei, Taiwan; Ming M. Deng, Taipei, Taiwan; Lein M. Chi, Taipei, Taiwan

[73] Assignee: Trend Micro, Incorporated, Cupertino, Calif.

[21] Appl. No.: 08/724,949

[22] Filed: Oct. 2, 1996

[51] Int. Cl.^{6} G06F 11/00

[52] U.S. Cl. 714/38

[58] Field of Search 395/183.14, 183.15, 395/183.13, 183.08, 183.09, 704, 186, 188.01, 187.01; 380/3, 4

[56] References Cited

U.S. PATENT DOCUMENTS

4,975,950	12/1990	Lentz	390/4
5,278,901	1/1994	Shieh et al.	380/4
5,319,776	6/1994	Hile et al.	395/575
5,321,840	6/1994	Ahlin et al.	395/700
5,408,642	4/1995	Mann	395/183.14
5,414,833	5/1995	Hershey et al.	395/575
5,440,723	8/1995	Arnold et al.	395/183.14
5,444,850	8/1995	Chang	395/200.1
5,448,668	9/1995	Perelson et al.	395/182.19
5,452,442	9/1995	Kephart	395/183.14
5,485,575	1/1996	Chess et al.	395/183.14
5,491,791	2/1996	Glowny et al.	395/183.13
5,511,163	4/1996	Lerche et al.	398/183.04
5,530,757	6/1996	Krawczyk	380/23
5,550,976	8/1996	Henderson et al.	395/200.06
5,550,984	8/1996	Gelb	395/200.17
5,649,095	7/1997	Cozza	395/183.15

FOREIGN PATENT DOCUMENTS

666671	8/1995	European Pat. Off.	H04L 29/06
6350784	6/1994	Japan	H04N 1/00
9322723	11/1993	WIPO	G06F 11/00

OTHER PUBLICATIONS

Malarky, Anti Virus comparative Review, Virus Bulletin, pp. 10–11, May 1996.

Meyer, "Possible Macro Virus Attacks and How to Prevent Them", Computers & Security, vol. 15, No. 7, pp. 595–625, 1996.

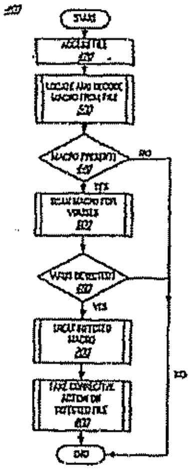
Primary Examiner—Joseph E. Palys

Attorney, Agent, or Firm—Skjerven, Morrill, MacPherson, Franklin & Friel LLP; Norman R. Klivans

[57] ABSTRACT

The detection and removal of viruses from macros is disclosed. A macro virus detection module includes a macro locating and decoding module, a macro virus scanning module, a macro treating module, a file treating module, and a virus information module. The macro locating and decoding module determines whether a targeted file includes a macro, and, where a macro is found, locates and decodes it to produce a decoded macro. The macro virus scanning module accesses the decoded macro and scans it to determine whether it includes any viruses. Unknown macro viruses are detected by the macro virus scanning module by obtaining comparison data that includes sets of instruction identifiers from the virus information module and determining whether the decoded macro includes a combination of suspect instructions which correspond to instruction identifiers. The macro treating module locates suspect instructions in the decoded macro using the comparison data and removes the suspect instructions to produce a treated macro. The file correcting module accesses a targeted file with an infected macro and replaces the infected macro with the treated macro produced by the macro treating module.

25 Claims, 7 Drawing Sheets



Defendant's Trial Ex.
DTX - 1022
Case No. 06-369 GMS

DEPOSITION EXHIBIT
8
Wallach
PENGAD 800-631-6989

FIG. 1

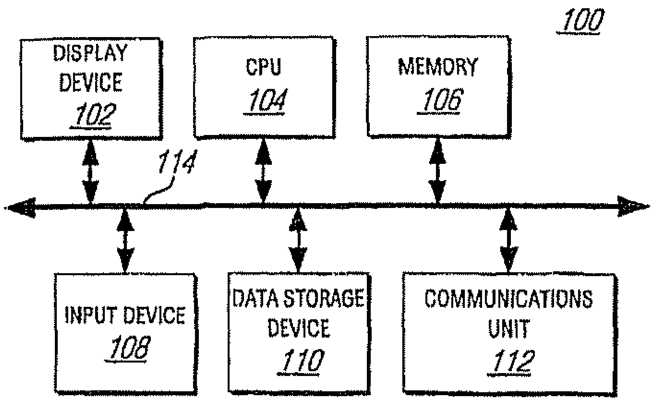


FIG. 2

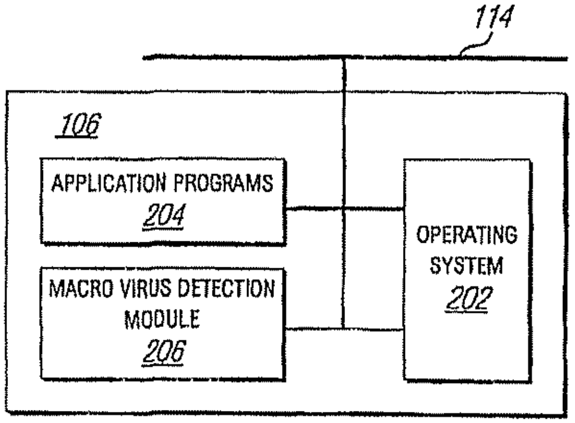
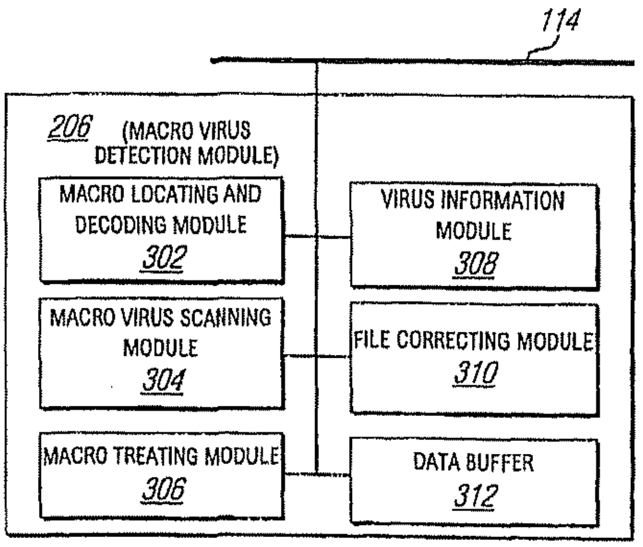


FIG. 3


114
206 (MACRO VIRUS DETECTION MODULE)
MACRO LOCATING AND DECODING MODULE 302
MACRO VIRUS SCANNING MODULE 304
MACRO TREATING MODULE 306
VIRUS INFORMATION MODULE 308
FILE CORRECTING MODULE 310
DATA BUFFER 312

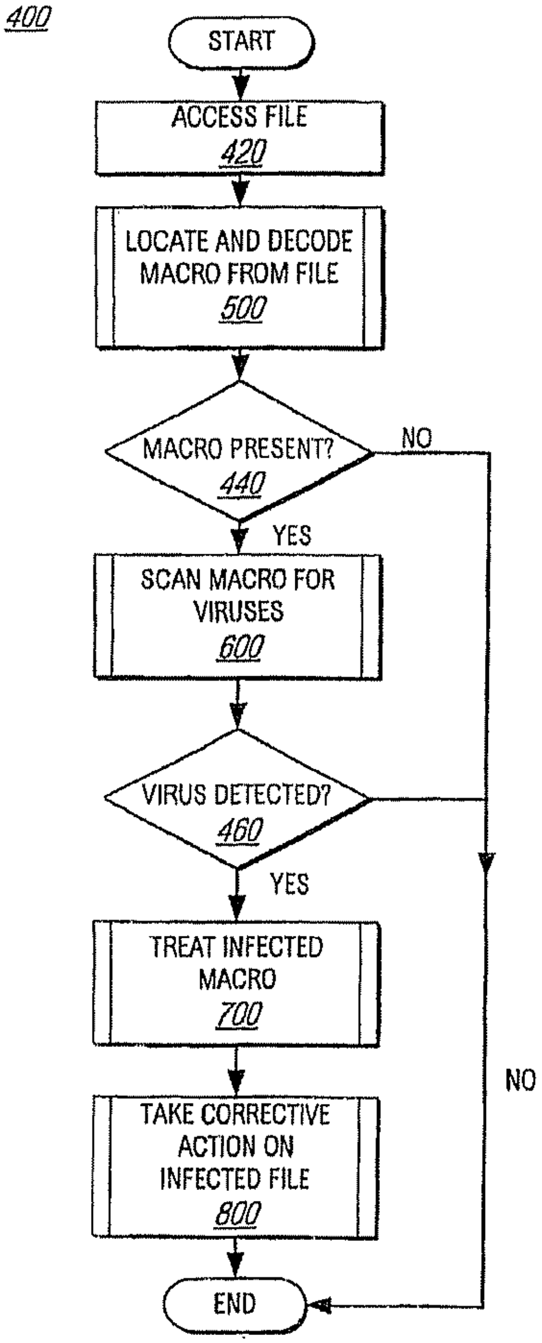

400
START
ACCESS FILE
420
LOCATE AND DECODE MACRO FROM FILE
500
MACRO PRESENT?
440
NO
YES
SCAN MACRO FOR VIRUSES
600
VIRUS DETECTED?
460
YES
TREAT INFECTED MACRO
700
NO
TAKE CORRECTIVE ACTION ON INFECTED FILE
800
END

FIG. 4

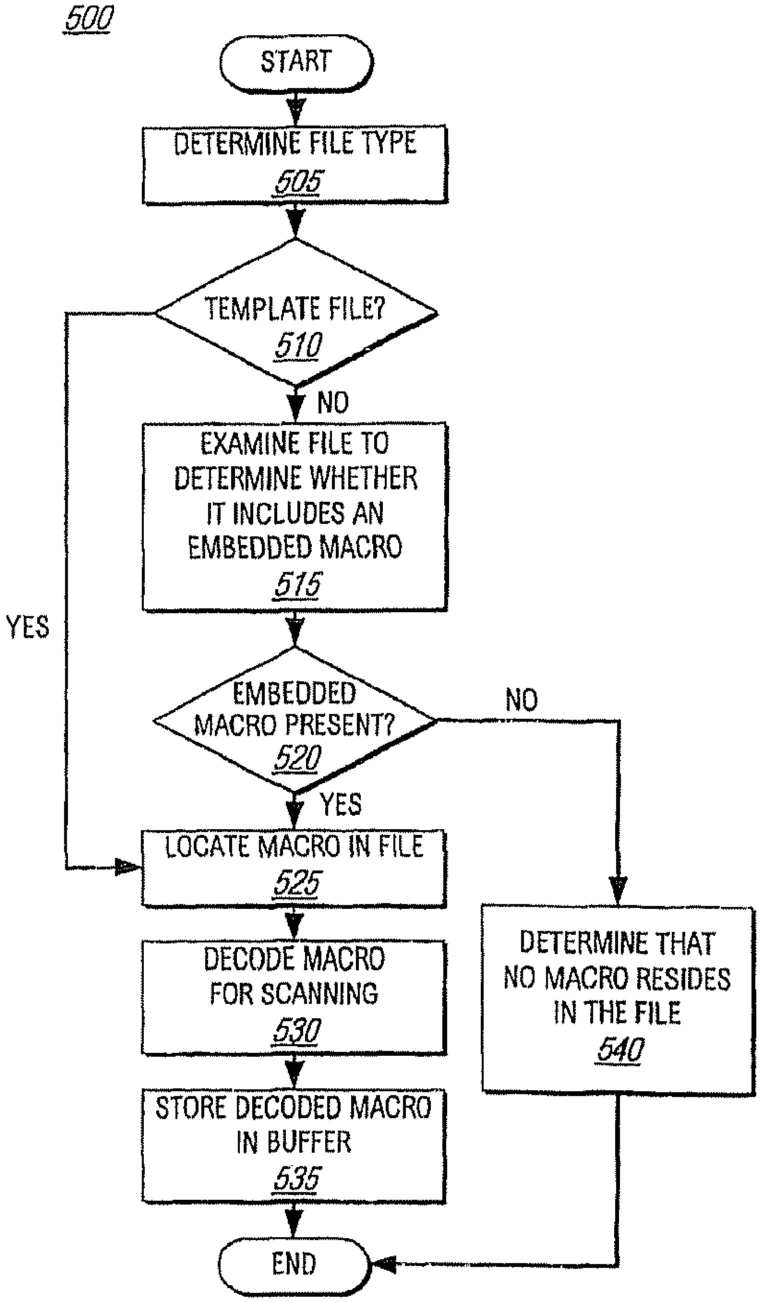

500
START
DETERMINE FILE TYPE
505
TEMPLATE FILE?
510
YES
NO
EXAMINE FILE TO DETERMINE WHETHER IT INCLUDES AN EMBEDDED MACRO
515
EMBEDDED MACRO PRESENT?
520
NO
YES
LOCATE MACRO IN FILE
525
DECODE MACRO FOR SCANNING
530
STORE DECODED MACRO IN BUFFER
535
DETERMINE THAT NO MACRO RESIDES IN THE FILE
540
END

FIG. 5

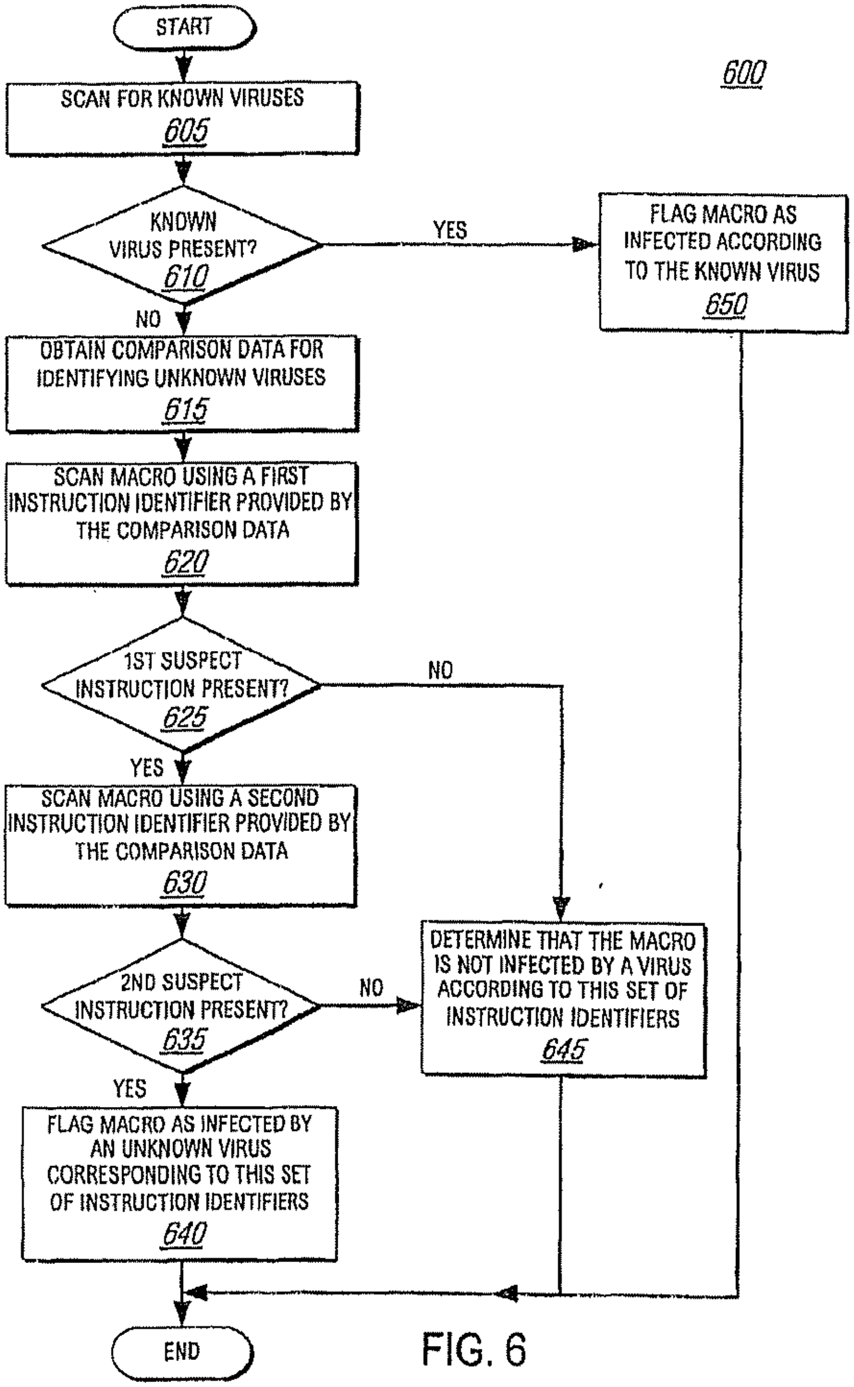

600
START
SCAN FOR KNOWN VIRUSES
605
KNOWN VIRUS PRESENT?
610
YES
FLAG MACRO AS INFECTED ACCORDING TO THE KNOWN VIRUS
650
NO
OBTAIN COMPARISON DATA FOR IDENTIFYING UNKNOWN VIRUSES
615
SCAN MACRO USING A FIRST INSTRUCTION IDENTIFIER PROVIDED BY THE COMPARISON DATA
620
1ST SUSPECT INSTRUCTION PRESENT?
625
NO
YES
SCAN MACRO USING A SECOND INSTRUCTION IDENTIFIER PROVIDED BY THE COMPARISON DATA
630
2ND SUSPECT INSTRUCTION PRESENT?
635
NO
DETERMINE THAT THE MACRO IS NOT INFECTED BY A VIRUS ACCORDING TO THIS SET OF INSTRUCTION IDENTIFIERS
645
YES
FLAG MACRO AS INFECTED BY AN UNKNOWN VIRUS CORRESPONDING TO THIS SET OF INSTRUCTION IDENTIFIERS
640
END

FIG. 6

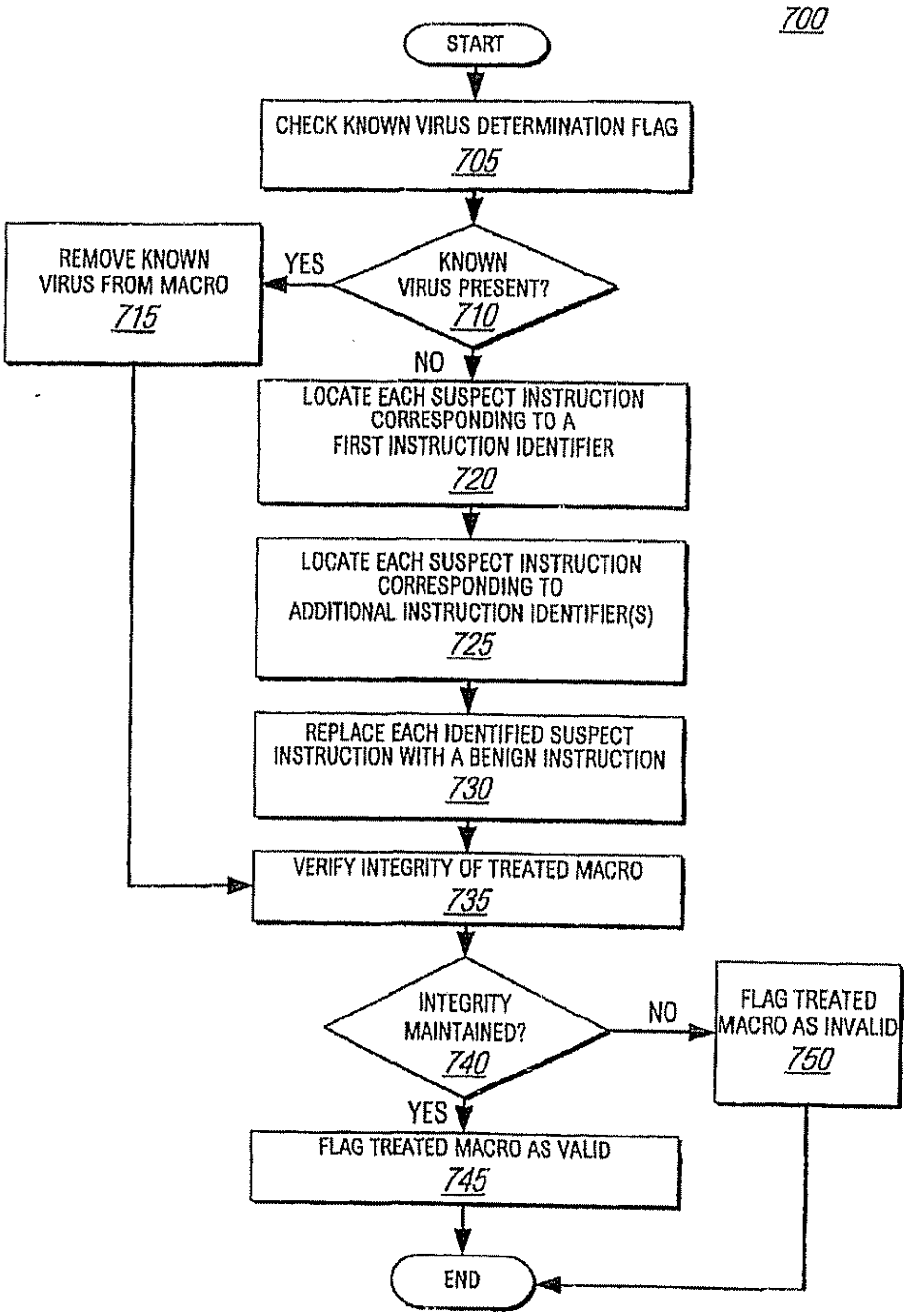

700
START
CHECK KNOWN VIRUS DETERMINATION FLAG
705
KNOWN VIRUS PRESENT?
710
YES
REMOVE KNOWN VIRUS FROM MACRO
715
NO
LOCATE EACH SUSPECT INSTRUCTION CORRESPONDING TO A FIRST INSTRUCTION IDENTIFIER
720
LOCATE EACH SUSPECT INSTRUCTION CORRESPONDING TO ADDITIONAL INSTRUCTION IDENTIFIER(S)
725
REPLACE EACH IDENTIFIED SUSPECT INSTRUCTION WITH A BENIGN INSTRUCTION
730
VERIFY INTEGRITY OF TREATED MACRO
735
INTEGRITY MAINTAINED?
740
NO
FLAG TREATED MACRO AS INVALID
750
YES
FLAG TREATED MACRO AS VALID
745
END

FIG. 7

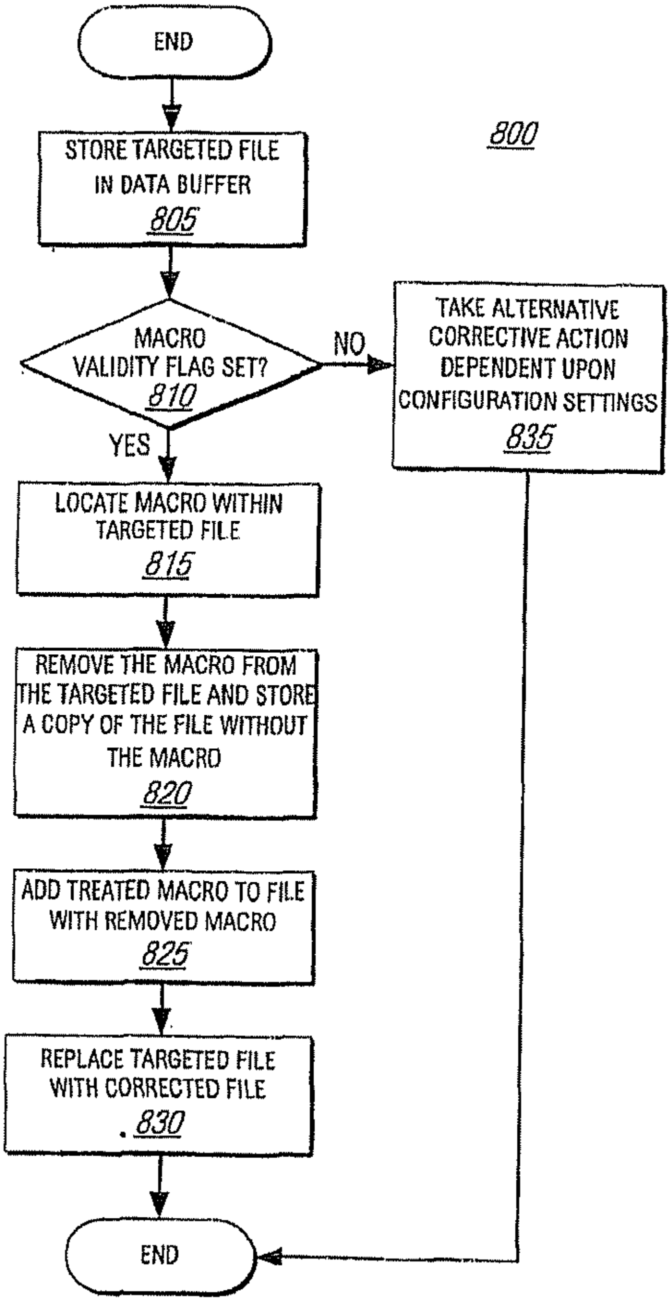
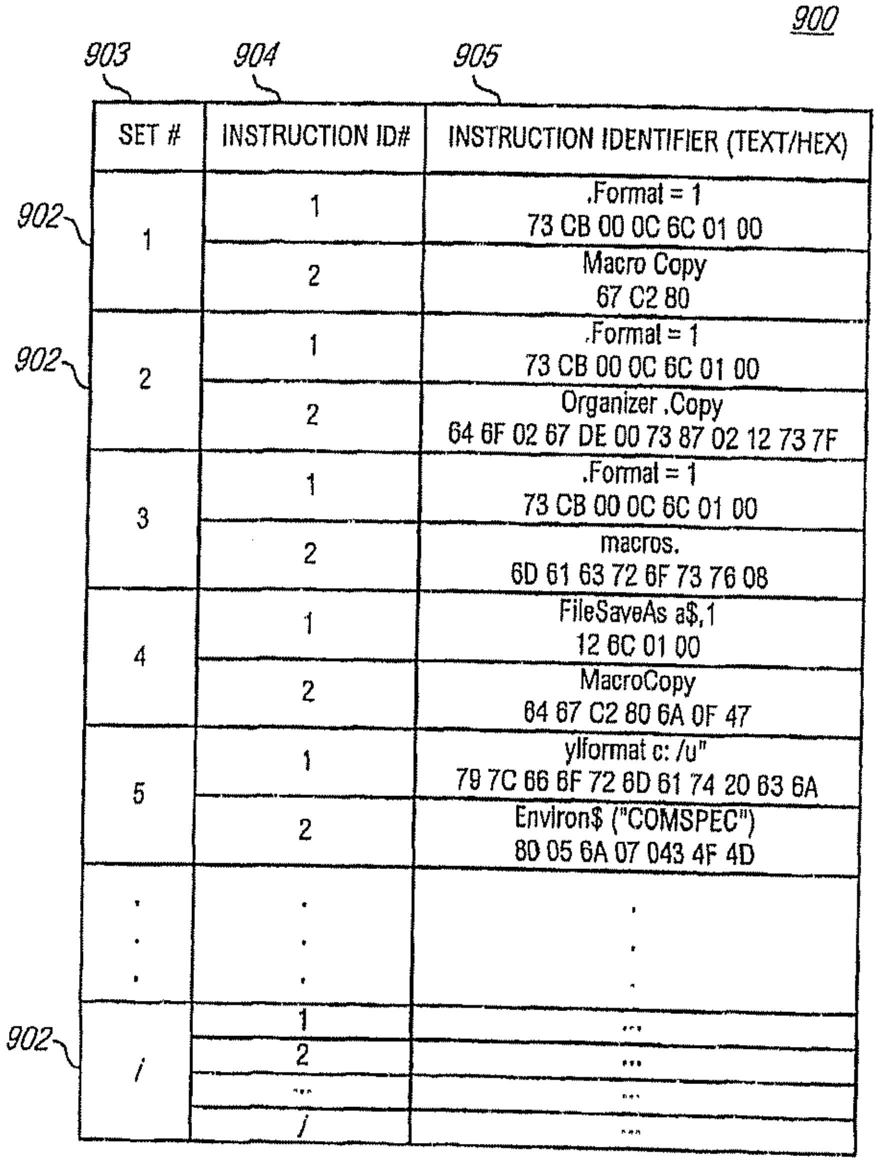

END
800
STORE TARGETED FILE IN DATA BUFFER 805
MACRO VALIDITY FLAG SET? 810
NO
YES
TAKE ALTERNATIVE CORRECTIVE ACTION DEPENDENT UPON CONFIGURATION SETTINGS 835
LOCATE MACRO WITHIN TARGETED FILE 815
REMOVE THE MACRO FROM THE TARGETED FILE AND STORE A COPY OF THE FILE WITHOUT THE MACRO 820
ADD TREATED MACRO TO FILE WITH REMOVED MACRO 825
REPLACE TARGETED FILE WITH CORRECTED FILE 830
END

FIG. 8


900
903
904
905
902
902
902

SET #	INSTRUCTION ID#	INSTRUCTION IDENTIFIER (TEXT/HEX)
1	1	.Format = 1 73 CB 00 0C 6C 01 00
	2	Macro Copy 67 C2 80
2	1	.Format = 1 73 CB 00 0C 6C 01 00
	2	Organizer .Copy 64 6F 02 67 DE 00 73 87 02 12 73 7F
3	1	.Format = 1 73 CB 00 0C 6C 01 00
	2	macros. 6D 61 63 72 6F 73 76 08
4	1	FileSaveAs a$,1 12 6C 01 00
	2	MacroCopy 64 67 C2 80 6A 0F 47
5	1	ylformat c: /u" 79 7C 66 6F 72 6D 61 74 20 63 6A
	2	Environ$ ("COMSPEC") 80 05 6A 07 043 4F 4D
. . .	. . .	. . .
i	1	...
	2	...
	...	...
	j	...

FIG. 9

SYSTEM, APPARATUS AND METHOD FOR THE DETECTION AND REMOVAL OF VIRUSES IN MACROS

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates generally to the detection of viruses in computer files and more particularly to the detection and removal of viruses in macros.

2. Description of the Related Art

The increasing use of computers and the increasing communication between vast numbers of computers has greatly facilitated and promoted the spread of computer viruses. Computer viruses are found in portions of code which are buried within computer programs. When programs that are infected with the virus are executed, the code portions are activated to provide unintended and sometimes damaging operations in the computer system.

Viruses are commonly detected using signature scanning techniques. Known viruses have extended strings or signatures which are like a fingerprint that can be used to detect the virus. In signature scanning an executable file sequence is scanned to see if it includes an extended string that matches a string that is known to be a virus. When the signature or string is found within the executable file, a positive virus determination is made. Since matching to known patterns is involved, the signature scan is virtually useless against viruses whose patterns have not been previously identified. Particularly, signature scanning completely fails to detect new, unknown virus strains, and does not adequately protect against mutating viruses, which intentionally assume various shapes and forms upon replication. Additionally, since executable files (e.g., those with extensions .exe or .com) are typically scanned, viruses which do not reside in such files are not examined and therefore are not detected with signature scanning.

Many application programs support the use of macros which are used to automatically perform long or repetitive sequences of actions. Macros are a series of instructions including menu selections, keystrokes and/or commands that are recorded and assigned a name or a key. Macros can be triggered by the application program such as by pressing the key or calling the macro name. Some macros are embedded within application data files, and thus may remain hidden from the user. Additionally, macros can be arranged to execute automatically, without user input. Thus, a macro which the user does not know about and which does not need manual triggering by the user may reside in files such as application data files. Certain viruses reside in macros and use macro instructions to perform unintended and sometimes damaging actions. These viruses are referred to as macro viruses. One problem with macro viruses is that they avoid executable file scanners since they typically do not reside in executable files. Additionally, macro viruses avoid detection since they can be hidden or embedded within files such as application data files. Furthermore, there are a tremendous number of computer users who know how to use macro programming languages so the number and variation of macro viruses is extremely high. Accordingly, even if signature scanning techniques were used to detect viruses in macros they would be ineffective since there are numerous unknown macro viruses. Additionally, even if a comprehensive signature scanner were available, it would quickly become obsolete because of the ongoing generation and production of new, unknown macro viruses.

Conventional virus treatment techniques are also inadequate for treating macros with viruses. These techniques look for particular, known viruses and apply a specific correction technique dependent upon the particular virus detected. Because of the vast array of unknown macro viruses, such techniques are not very effective in the treatment of macros which are infected by viruses. Even if unknown macros were detected, merely deleting the file with the infected macro is not an attractive solution, since infected macros often include legitimate operations that the user may want to retain. Thus, there is a need to selectively remove viruses, particularly unknown viruses, from macros to provide a clean, corrected file which can be subsequently used.

Another problem is that viruses, and therefore, the information that is needed to detect them, are in a constant state of change. Thus, a virus detection method and apparatus which facilitates easy updates of virus detection information is needed. Particularly required is easily modified unknown macro virus detection information.

Thus, there is a need for the detection of viruses which reside in macros. Moreover, there is a need for detecting unknown macro viruses, for cleaning macro viruses selectively, and for conveniently updating macro virus detection information.

BRIEF SUMMARY OF THE INVENTION

The present invention overcomes the limitations and shortcomings of the prior art with systems, apparatuses and methods for detecting and removing viruses from macros.

In accordance with the present invention, a macro virus detection module includes a macro locating and decoding module, a macro virus scanning module, a macro treating module, a virus information module, a file correcting module, and a data buffer. A file is targeted for virus detection according to the configuration settings of the macro virus detection module and is copied into the data buffer for analysis. The macro locating and decoding module examines the file to determine whether it is a template file. If it is determined that the file is a template file, any macros within the template file are located and decoded. If the target file is not a template file, then the macro locating and decoding module examines the target file to determine whether it includes any embedded macros that are then located and decoded. The decoded macros are stored in the data buffer.

The macro scanning module is in communication with the macro locating and decoding module and the data buffer and thus accesses the decoded macros for virus scanning. The macro virus scanning module is also in communication with the macro virus information module. The macro virus information module includes information that is used by the macro virus scanning module to detect both known and unknown viruses in macros. First, a decoded macro is scanned for known viruses. If a known virus is detected, then the decoded macro is flagged as infected. The decoded macro, the flag and information associating the decoded macro to the known virus that was detected in the decoded macro are stored in the data buffer. Thus, the infected macro and the file that the infected macro resides in can be properly treated and corrected by the macro treating module and the file correcting module.

If a known virus is not detected, then the macro virus scanning module determines whether the decoded macro includes an unknown virus. The macro virus scanning module detects unknown macro viruses using comparison data which is stored in the virus information module. The comparison data includes information that is used to detect combinations of suspect instructions in macros. An exemplary set of comparison data includes first and second suspect instruction identifiers. The macro virus scanning module determines that the macro includes a virus if it is determined that the macro includes both the first and second suspect instructions. If an unknown macro virus is detected, then the macro is flagged as infected according to the set of instruction identifiers that resulted in the positive unknown virus detection. As with the detection of a known virus, the information which resulted in the positive detection is stored in the data buffer along with the infected macro so that the macro treating module and the file correcting module can properly treat and correct the infected macro. Since combinations of suspect instructions in lieu of a particular sequential signature are sought, unknown viruses are detected by the macro virus scanning module. The information for detecting known and unknown viruses resides in a separate module and thus is easily updated.

The macro treating module is in communication with the macro virus scanning module and the data buffer and thereby accesses information about detected viruses. The macro treating module removes detected viruses from macros to provide a cleaned or sanitized macro so that the file which includes the infected macro can be repaired or otherwise corrected by the file treating module. The macro treating module accesses the decoded macro in the data buffer and determines whether it has been flagged as infected with a known virus. If the macro is flagged as infected by a known virus, then the known virus is removed from the macro. If the macro is not infected by a known virus, then the macro treating module treats the macro using the set of instruction identifiers that was used to detect an unknown virus in the macro. The macro treating module receives decoded macros and information regarding the presence of viruses from the macro virus scanning module and the data buffer. The suspect instructions within the decoded macro are identified and located using the instruction identifiers. The suspect instructions are then removed from the infected macro, preferably by replacement with benign instructions, to provide a treated macro which has been cleaned or sanitized. The treated macro is stored in the data buffer for access by the file correcting module. The integrity of the treated macro is verified and the validity of the treated macro is flagged accordingly. If macro integrity is maintained upon completion of macro treatment, then the validity flag is set. If integrity is not maintained, the flag is not set.

The file correcting module is in communication with the macro locating and decoding module, the macro virus scanning module, the macro treating module, the data buffer and the virus information module. Information about the treated macro and the targeted file that includes an infected macro are accessed in the data buffer. The file correcting module accesses the targeted file in its original form and stores a copy of the targeted file in the data buffer. The copy of the targeted file includes the infected macro. If the macro validity flag is not set, then the treated macro is not used to replace the infected macro and alternative corrective action is taken, such as deleting the targeted file, notifying the user about the presence of a file with an infected macro, or removing the infected macro from the targeted file and replacing the targeted file with the version without a macro. If the macro validity flag is set, then the file correcting module corrects the targeted file by replacing the infected macro with the treated macro. To replace the infected macro, the file correcting module locates and removes the infected macro from the targeted file so that a version of the targeted file without the macro is stored in the data buffer. The treated

macro is then added to the version of the targeted file without the macro to provide a corrected file. The corrected file is used to replace the targeted file (in its original location). Thus, unknown viruses are removed from macros and the files which include such macros are corrected so that their legitimate functions are retained.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other more detailed and specific features of the present invention are more fully disclosed in the following specification, reference being had to the accompanying drawings, in which:

FIG. 1 is a block diagram illustrating a computer system including a macro virus detection apparatus in accordance with the present invention.

FIG. 2 is a block diagram illustrating a preferred embodiment of a memory in accordance with the present invention.

FIG. 3 is a block diagram illustrating a preferred embodiment of a macro virus detection module in accordance with the present invention.

FIG. 4 is a flow chart illustrating a preferred method of detecting and correcting viruses in macros in accordance with the present invention.

FIG. 5 is a flow chart illustrating a preferred method of locating and decoding macros in accordance with the present invention.

FIG. 6 is a flow chart illustrating a preferred method of scanning macros for viruses in accordance with the present invention.

FIG. 7 is a flow chart illustrating a preferred method of treating macros in accordance with the present invention.

FIG. 8 is a flow chart illustrating a preferred method of correcting files in accordance with the present invention.

FIG. 9 is a table including exemplary sets of comparison data used in the detection of macro viruses.

DETAILED DESCRIPTION OF THE INVENTION

Referring now to FIG. 1, a computer system **100** constructed in accordance with an embodiment of the present invention comprises a central processing unit (CPU) **104**, a display device **102**, a memory **106**, an input device **108**, a data storage device **110**, and a communications link **112**. The CPU **104** is coupled by a bus **114** to the display device **102**, memory **106**, input device **108**, data storage device **110** and communications unit **112** in a conventional architecture such as a von Nuemann architecture such as provided in a personal computer. The CPU **104** is preferably a microprocessor such as a Pentium as provided by Intel, Inc. of Santa Clara, Calif.; the display device **102** is preferably a video monitor; the memory **106** is preferably random access memory (RAM); the input device **108** preferably comprises a keyboard and mouse; the data storage device **110** is preferably a hard disk; and the communications unit **112** is a device, such as a modem, for facilitating communication with other systems.

It is understood that a variety of alternative computer system configurations are available and that the present invention is independent of their use. For example, alternative processors may be used for the CPU **104** such as those provided by Motorola, Inc., and the memory **106** may be read only memory (ROM) or a combination of RAM and ROM. Additionally, the system **100** may be connected to other computer systems such as through a network interface (not shown). It is also understood that the computer system 100 may be a minicomputer or mainframe computer.

The CPU 104, as directed by instructions received from the memory 106 as configured in accordance with the present invention, provides signals for accessing computer files, determining whether they include macros, locating the macros and scanning them to determine whether viruses including unknown viruses are present, and taking corrective action when such viruses are detected.

Referring now to FIG. 2, a preferred embodiment of the memory 106 as configured in accordance with the present invention is shown in more detail. The memory 106 includes an operating system 202, application programs 204 and a macro virus detection module 206.

The operating system 202 is preferably a conventional one for a personal computer such as WINDOWS 3.1 as provided by Microsoft, Inc. of Redmond, Wash. Any of the wide variety of application programs 204 may be included in the memory 106 such as word processing, spreadsheet and drawing programs. For example, the memory 106 may include Microsoft WORD as a word processing application and Microsoft EXCEL as a spreadsheet application The application programs 204 typically create application data files. For example, WORD generates data files which typically use the file extension .DOC. Typical application programs 204 accommodate macros which allow sequential operations without necessitating repetitive user input. Conventional macros include various instructions including those for relatively simple operations such as keystrokes as well as those for operations such us opening, copying and deleting files. Sometimes, the macro instructions invoke the operating system (or DOS shell) to execute lower level commands like FORMAT. The instructions which may be used by macros are usually dictated by the application programs 204 which support macro programming languages. For example, macros for WORD files may be written using the WordBasic programming language.

Various operating systems 202 may be alternatively used in accordance with the present invention, such as OS/2 as provided by IBM, Inc. Also, various application programs 204 may be used. Although certain embodiments of the present invention describe the detection of macro viruses which use WordBasic commands in WORD application data files, the artisan will understand that the present invention applies to such alternative operating systems 202 and application programs 204.

The macro virus detection module 206 includes routines for the accessing files, determining whether such files include macros, scanning the macros to determine whether they contain viruses, treating macros which are found to contain viruses and correcting files with infected macros. The macro virus detection module 206 operates in conjunction with the operating system 202 and the application programs 204. The macro virus detection module 206 is typically implemented in software, but can also be implemented in hardware or firmware. Although the macro virus detection module 206 is preferably separate from the operating system 202 and application programs 204 as shown, the macro virus detection module 206 can alternatively be integrated into the operating system 202 or application programs 204 to perform similar functions of virus detection and correction.

Referring now to FIG. 3, a preferred embodiment of the macro virus detection module 206 includes a macro locating and decoding module 302, a macro virus scanning module 304, a macro treating module 306, and a file correcting module 310. Additionally, a virus information module 308 provides comparison data for the detection of viruses in macros and the treatment of macros with viruses, and the data buffer 312 provides for the storage of information which is used in the detection and correction of macro viruses. Although the data buffer 312 is shown as a single module which includes several separate storage locations, a plurality of separate data buffers may be used for the various described functions of the data buffer 312.

The macro virus detection module 206 accesses targeted files and determines whether they includes macros. Files are accessed dependent upon the configuration settings of the module 302 which are preset or determined by the user. For example, a system user may target a single file for analysis. Alternatively, groups of files may be targeted such as the files corresponding to a selected application program 204 or all files within selected directories or storage areas. Various events may initiate the analysis of files. For example, the user may initiate a virus scan, the analysis may be initiated any time certain application files are opened, or a complete analysis may be undertaken at every nth boot up of the system 100 or at periodic interval. Preferably, the macro locating and decoding module 302 is arranged to access any files which may contain a macro virus and to access such files prior to the launching of an application program 204 and accordingly prior to the opening of any application data files. This is advantageous because some macro viruses operate automatically upon a launching of the relevant application program and, accordingly, may require detection prior to user initiation of a scan.

Each targeted file is accessed and stored in the data buffer 312 for analysis by the macro virus detection module 206. While, for ease of understanding, the analysis of a single file is described in connection with certain functions of the various modules 302, 304, 306, 308, 310, 312 several files can be concurrently or sequentially analyzed in accordance with the present invention.

The macro locating and decoding module 302 examines targeted files to determine whether they are of a type that may include macros, examines targeted files to determine whether they include embedded macros, and locates and decodes macros within targeted files.

The macro locating and decoding module 302 is in communication with the data buffer 312 and thus accesses a targeted file for analysis. Macros are found in template files and embedded within application data files. The macro locating and decoding module 302 initially determines whether the targeted file is a template file. This determination is made by checking the file extension. For example, if the file is associated with the WORD application program 204, then the file is checked for the extension DOT. The DOT extension indicates that the file is a template file.

If the targeted file is not determined to be a template file, it may still include an embedded macro. For example, an application data file such as a WORD file with a .DOC extension may include an embedded macro. The macro locating and decoding module 302 accesses the targeted file stored in the data buffer 312, and determines whether its formatting indicates an embedded macro. The formatting fields vary according to the conventional rules of each application program 204 and are supplied by application program 204 manufacturers

The macro is located within the targeted file once the determination is made that the targeted file includes a macro, whether it is embedded, provided in a template file or provided in another file type that can support macros. The macro locating and decoding module 302 is in communication with the operating system 202. The operating system includes information sharing resources such as Object Linking and Embedding (OLE, or OLE2) as provided in the Windows 3.1 operating system. The information sharing resources provide details regarding the structure of files such as application files, so that objects which are embedded may be located within the files. The information sharing resource commands vary dependent upon the operating system 202, but generally simple commands such as those for opening objects, seeking particular streams, and reading and writing to files are provided. Conventional programming techniques can be used to implement the information sharing resources in locating and decoding macros. After the macro is located, the macro locating and decoding module 302 decodes the macro so that it can be scanned for viruses. The information sharing resources of the operating system 202 are used to decode the macros into coherent information and ASCII conversion is used to convert the decoded macro into a form which is suitable for scanning. The decoded macro is stored in the data buffer 312. Additionally information associating the decoded macro to the targeted file from which it was derived are stored in the data buffer 312.

The macro virus scanning module 304 is in communication with the macro locating and decoding module 302 and the data buffer 312, and, accordingly, the macro locating and decoding module 302 provides the decoded macro to the macro scanning module 304. A preferred method of locating and decoding macros which is used by the macro locating and decoding module 302 is described in further detail with reference to FIG. 5.

The macro virus scanning module 304 includes routines for scanning the decoded macros to detect known and unknown viruses based upon comparison of the decoded macros to data from the virus information module 308. The macro virus scanning module 304 may be configured to provide various modes of macro virus detection. For example, it may be configured to detect only certain classes of viruses to facilitate shorter scanning periods; known, unknown, or both types of viruses may be detected; an alarm may be provided upon the first detection of a virus; or a scan of several targeted files may be completed prior to the indication that a virus has been detected.

The macro virus scanning module 304 accesses decoded macros in the data buffer 312, scans the decoded macros for known viruses and, where known viruses are not found, scans the decoded macros for unknown viruses. To scan for known viruses, the macro virus scanning module 304 uses signature scanning techniques. The virus scanning module 304 is in communication with the virus information module 308. The virus information module 308 includes information for detecting known viruses. For example, the virus information module includes strings of data, or signatures, for identifying known viruses. The virus scanning module 304 accesses a decoded macro in the data buffer 312 and scans the decoded macro to determine whether it includes any known virus signatures. A state machine or similar techniques can be used to conduct the scanning. If a known virus signature is found in the decoded macro, then the macro virus scanning module 304 identifies the decoded macro as infected according to the known virus and stores information associating the decoded macro to the known virus in the data buffer 312 so that other modules such as the macro treating module 306 can subsequently treat the infected macro accordingly.

If known viruses are not detected, the macro scanning module 304, scans for unknown viruses in the decoded

macros. As indicated above, application programs 204 often include programming languages, such as WordBasic, which include instructions that are used by macros for various operations. Macro viruses use the various operations and instructions to perform undesired and often damaging actions.

Typical application programs 204 support macros by providing them in template files. Template files include stylistic and other settings such as word processing settings. Additionally, template files can include macros. Typically, a global template file provides the settings and macros for the data files. For example, for Microsoft WORD, the global settings and pool of macros reside in the template file NORMAL.DOT. When an application program 204 opens a data file, it usually opens the global template file first, loads the global settings and macros, and then opens the data file. Typical data files are formatted to indicate to the application program 204 that they do not include embedded macros. However, a data file can be formatted to indicate to the application program 204 that it does include a template file.

Certain macro viruses cause infected document files to be saved in template formats but maintain a document or data file extension (.DOC) to remain undetected. Thus, infected macros may be embedded within a document that appears to merely be an application data file. Some such macros include commands such as "AutoOpen", "AutoExec" or "AutoClose" which cause the macro to be executed when the data file is opened. Thus, a user can attempt to open what appears to be a regular data file but, in doing so, will be causing an embedded macro to execute automatically. Macro viruses also replicate themselves in other files. For example, a macro virus will often copy itself into a data file and format the data file to indicate a template format while maintaining a regular file extension for the data file. The file which is attacked by the macro virus may have its formatting changed directly, or, alternatively, the macro virus may save the attacked data file with the updated format information. The infected macros may also be copied into the global template and thereby spread to other files as they are opened.

The macro virus scanning module 304 includes routines to detect macro instruction combinations which are very likely to be used by macro viruses, or, in other words, combinations of suspect instructions. One combination of suspect instructions that is detected by the macro virus scanning module 304 is a macro enablement instruction and a macro reproduction instruction. A macro enablement instruction is one which allows the formatting of a file to be set to indicate that the file include a macro for execution. For example, the file formatting may be set to indicate a template file so that an application program 204 causes the template file to be executed when the file is opened. A macro reproduction instruction is one which allows the macro virus to be replicated. The combination of a macro enablement instruction and a macro reproduction instruction indicates a macro virus since such instructions allow replication and execution of a macro in a destination file, two common characteristics of macro viruses.

To identify suspect instruction combinations, the macro virus scanning module 304 accesses comparison data from the virus information module 308. The comparison data includes sets of instruction identifiers which are used to identify combinations of suspect instructions in the decoded macro. An exemplary set of instruction identifiers includes first and second suspect instruction identifiers. The instruction identifiers are binary strings and the decoded macros are scanned to determine whether they include the binary strings and, accordingly, the suspect instructions. If it is determined that the macro includes the combination of suspect instructions defined and identified by the set of instruction identifiers, then it is determined that the macro is infected by an unknown virus corresponding to that set of data. The macro virus scanning module 304 flags the decoded macro as infected with an unknown virus and stores information associating the decoded macro to the set of instruction identifiers that resulted in a positive unknown virus detection in the data buffer 312 so that other modules such as the macro treating module 306 can treat the infected macro accordingly. Combinations of suspect instructions which are detected by the macro virus scanning module 304, including those for macro virus enablement and macro virus reproduction, are described in further detail with reference to the flow chart of FIG. 6.

Preferably, the virus information module 308 is separate from the other modules 302, 304, 306, 310, 312 in the macro virus detection module 206. Thus, the information for the detection of macro viruses may be easily updated. For example, the virus information 308 may be updated by copying new information obtained from sources such as a floppy disk. Alternatively, the new information may be downloaded from internet resources which may be accessed through the communications unit 112 of the computer system 100, or through network links (not shown). The separate virus information module 308 facilitates more efficient information transfer, is easier to update and, thus, provides better protection of the system 100 from viruses, including those which are unknown.

The macro treating module 306 is in communication with the data buffer 312 and the macro virus scanning module 304 and accordingly receives information regarding the detection of viruses within the decoded macros from targeted files. The macro treating module 306 includes routines for checking the status of decoded macros to determine whether the macro virus scanning module 304 detected a known or unknown virus in the decoded macros, for removing macro viruses from the decoded macros and for verifying the integrity of treated macros.

The macro treating module 306 accesses a decoded macro in the data buffer 312 and checks the status of the decoded macro to determine whether the macro virus scanning module 304 detected a known virus. The status is provided by information in the data buffer 312 such as a status flag. Additionally, the data buffer 312 includes information associating the decoded macro to the known virus which infects it. As indicated, this information is provided from the macro virus scanning module 304. Alternatively, the macro virus scanning module 304 can store the appropriate information and communicate directly with the macro treating module 306. If the known virus determination flag is set, the macro treating module uses the information about the known virus to remove the known virus from the decoded macro. Preferably, known macro virus is selectively removed from the decoded macro and is replaced with benign instructions so that the remaining portions of the macro are retained for future use. The treated macro is separately stored in the data buffer 312 by the macro treating module 306. The integrity of the treated macro is then ascertained by the macro treating module and, where such integrity is maintained the treated macro is flagged as valid. If the integrity of the treated macro is not maintained, then the treated macro is flagged as invalid. The integrity of the macro can be checked by ascertaining whether the remaining instructions are intact and, additionally, by ascertaining whether the sequential linking of the instructions remains intact. Macro integrity verification allows other modules such as the file correcting

module 310 to determine alternative remedies such as whether to replace an infected macro with a treated macro or to merely delete an infected macro from a targetted file.

If the macro virus scanning module 304 detected an unknown virus, then the macro treating module 306 treats the macro to remove the effects of the unknown virus. Similar to the known virus treatment protocol, the treating module 306 communicates with the data buffer 312 to determine whether an unknown virus was detected and, if so, to identify the instruction identifier set which resulted in a positive virus determination. Preferably, the set of suspect instruction identifiers that resulted in the detection of an unknown virus in the macro are used to correct the macro. Each instruction identifier is correlated to one or more suspect instructions so that such instructions may be identified and removed from the macro. The virus correction 306 locates and decodes the macro for correction or, alternatively, accesses macros which were previously decoded by the macro virus scanning module 304. Again, the suspect instructions are preferably removed from the decoded macro and replaced with benign instructions. The treated macro is stored in the data buffer 312 for access by other modules such as the file correcting module 310. Similar to the known virus treatment protocol, the integrity of the treated macro is verified and the macro is flagged accordingly. A preferred macro treating routine is described in more detail with reference to FIG. 7.

The file correcting module 310 is in communication with the data buffer 312 and other modules such as the macro treating module 306 and thereby receives information about the viruses which were detected in macros from targeted files. The file correcting module 310 includes routines for providing remedial action where infected files are indicated. The routines in the correcting module 306 may be configured to take various forms of remedial action automatically or only upon user authorization. For example, the file correcting module 310 may copy a targeted file with an infected macro, replace the infected macro with a treated macro, and then replace the targeted file with a corrected file without informing the user. The file correcting module 310 can also be configured to prompt the user at various correction stages whether the user wishes to proceed. This, of course, may be undertaken interactively using the input 108 and display devices 102 of the computer system 100. For example, the correcting module 310 may indicate to the user that a certain type of virus or unknown virus was detected in a macro from a targeted file. The user may then be asked whether he or she wishes to replace the targeted file with a corrected file. The artisan will recognize the various ways that the correcting module 306 may be configured and that the user may be prompted at various stages.

The file correcting module 310 communicates with the data buffer 312 which indicates targeted files that include infected macros. The file correcting module 310 accesses a targeted file that includes an infected macro and stores a copy of the targeted file in the data buffer 312 so that the file can be repaired. As described regarding the macro locating and decoding module 302, the macro virus scanning module 304, and the macro treating module 306 above, the data buffer 312 includes information about the targeted file such as an association of the targeted file to a treated macro, the validity of the treated macro, and information about the type of virus that was detected. The file correcting module 310 checks the macro validity flag in the data buffer 312 to determine whether the macro treating module 306 was able to remove the detected virus and maintain the integrity of the treated macro. If the integrity of the treated macro was maintained, then the file correcting module 310 repairs the targeted file in the data buffer 312 by replacing its infected macro with a treated macro. First, the viral macro is located within the infected file. This can be done by communication with the macro locating and decoding module 302 or may be independently done by the file correcting module 310, which like the locating and decoding module 302 can access the information sharing resources of the operating system 202 to locate the macro. After the macro is located within the infected targeted file, the macro is removed and a copy of the targeted file without the macro is stored in the data buffer 312. Then, the treated macro is added to the version of the targeted file that does not include the macro to provide a corrected file. The corrected file is used to replace the targeted file in its original location. This may be a direct replacement of the targeted file with the corrected file. Alternatively, the targeted file can be deleted or overwritten and the corrected file can be stored in a separate location.

Preferably a treated macro which corresponds to the targeted file which is flagged as invalid is not used to replace the infected macro. In such a situation, various alternative corrective actions may be taken by the file correcting module 310. For example, the user could be notified that the targeted file includes a virus, the infected macro could be removed from the infected macro without replacement, or the targeted file could be deleted.

Referring now to FIG. 4, a flow chart of a preferred method 400 of unknown virus detection in macros is shown. First, a file which is targeted for virus detection is accessed 420. Typically, targeted files will reside in memory 106 and such targeted files may be copied to the data buffer 312 so that the macro virus detection module 206 can detect and remove macro viruses from them. Although the processing of a single targeted file is described in detail, numerous files can be accessed and tested with the present invention. The timing and scope of file access depends upon how the macro virus detection module 206 is configured as explained in the description of the macro virus detection module 206 above. Specifically, various files may be targeted and the detection of viruses may be triggered based upon conditions which can be selected by the user. Preferably, though, the macro virus detection module 206 is configured to scan relevant application program 204 files including application data files without necessitating the launching of the application program 204 so that macro viruses which operate upon application program 204 launching can be detected and removed.

After accessing the targeted file and providing it in the data buffer 312, the macro locating and decoding module 302 determines whether any macros are in the targeted file and proceeds to locate and decode them. A preferred method 500 of locating and decoding a file is described in further detail with reference to FIG. 5. Next, it is determined 440 whether a macro was found by the locating and decoding module 302. If it is determined 440 that a macro is present in the targeted file, the macro is scanned 600 by the macro virus scanning module 304 to determine whether it is infected by a macro virus. If it is determined 440 that a macro is not present in the targeted file, then the method of macro virus detection ends. A preferred method of file scanning 600 is described in further detail with reference to FIG. 6. If a virus is detected 460 in the scan, then the infected macro is treated 700 by the macro treating module 306. If a virus is not detected 460 in the scan, then the method of virus detection ends. A preferred method of macro treating is described in further detail with reference to FIG. 7. After the macro is treated 700, corrective action 800 is taken on the infected targeted file after which the preferred method of macro virus detection ends. A preferred method of file correcting is described in further detail with reference to FIG. 8.

Referring now to FIG. 5, a flow chart of a preferred method 500 of macro locating and decoding in accordance with the present invention is shown. Various files may be targeted including application data files and template files. The macro locating and decoding module 302 includes routines which initially determine whether the targeted file includes a macro by determining in step 505 what type of file it is. This determination is made by checking the file extension of the targeted file. Through examination of the file type, it can be determined whether the targeted file is a template file. If the file is a template file, then the macro locating and decoding module 302 does not have to determine whether the targeted file includes an embedded macro. Thus, if it is determined in step 510 that the file is a template file, any macros within it are located 525 and decoded 530 so that they can be scanned for viruses by the macro virus scanning module 304. Any decoded macros are stored 535 in the data buffer 312 for access by the macro virus scanning module 304.

If it is determined in step 510 that the file is not a template file or other type of file that may include a macro, then in step 515 the targeted file is examined to determine whether it includes an embedded macro. If there are no embedded macros it is determined in step 540 that the targeted file does not include a macro and the preferred method ends. The determination of whether a file includes an embedded macro is performed by checking the file formatting. For example, one file format allows an application data file to indicate to an application program 204 that it includes a template file even where the file extension indicates otherwise. If the file format indicates a template file is included, then the targeted file may include an embedded macro. In step 520, if it is determined that the file does not include an embedded macro, then it is determined in step 540 that a macro does not reside in the targeted file and the preferred method 500 ends.

If it is determined in step 520 that the targeted file includes an embedded macro or in step 510 that the targeted file is a template file with a macro, the macro is located in step 525 and decoded in step 530. Preferably, the locating and decoding module 302 includes routines which use the operating system 202 information sharing resources such as Object Linking and Embedding (OLE, or OLE2) as provided in the Windows 3.1 operating system. As described regarding the locating and decoding module 302 above, the information sharing resources include instructions which provide details regarding the structure of files, such as application files and template files, so that objects which are integrated into files may be located and decoded. Conventional programming techniques can be used to implement the information sharing resources to locate and decode macros. After the macros are located and decoded, they are converted to binary code (e.g. by ASCII conversion) and in step 535 they are stored in the data buffer 312 so that can be scanned for viruses. In addition to storing the decoded macros, the macro locating and decoding module 302 maintains and stores an association between the decoded macro and the targeted file so that other modules such as the macro virus scanning module 304, the macro treating module 306 and the file correcting module 310 can properly scan, treat and correct targeted files. After the macros are decoded and stored in step 535, the preferred method 500 of locating and decoding ends.

Referring now to the flow chart of FIG. 6, a preferred method 600 for detecting viruses in macros in accordance with the present invention is shown. The macro virus scanning module 304 includes routines which access the decoded macro information provided by the locating and decoding module 302 and compare the decoded macro information to virus information 308 to detect the presence of a virus.

In a first step 605, the decoded macro is scanned for known viruses. To scan for known viruses, the macro virus scanning module 304 uses signature scanning techniques. The macro virus scanning module 304 accesses decoded macros in the data buffer 312 and determines whether they include known virus signatures that are provided by the virus information module. In step 610 it is determined whether the decoded macro includes a known virus based upon the known virus scanning step 605, and, if there are any known viruses present, in step 615 the macro virus scanning module flags the macro as infected according to the known virus and stores information associating the decoded macro to the known virus in the data buffer 312.

If in step 610 it is determined that known viruses were not detected in the scanning step 605, the macro virus scanning module 304 scans for unknown viruses in the decoded macros. In step 615, the macro virus scanning module 304 obtains a set of instruction identifiers for detecting unknown viruses. The macro scanning module 304 detects instructions which are likely to be used by macro viruses. These instructions are also referred to as suspect instructions. Certain combinations of suspect instructions are very likely to be used macro viruses. By looking for combinations of suspect instructions, false positive virus detection is avoided since a macro with two (or more) different suspect instructions is very likely to be infected.

As indicated in the description of the macro virus scanning module 304 above, typical application programs 204 provide macros in template files. Typically, application data files use a global template file but application data files can also be formatted to indicate that they include embedded macros. For example, WORD files can be saved in the .DOT format to indicate inclusion of a template file. Many macro viruses cause infected document files to be saved in template formats (.DOT) but maintain a document or data file extension (.DOC) to remain undetected. Thus, infected macros may be embedded within a document that appears to merely be an application data file. Macro viruses also replicate themselves in other files. For example, a macro virus will often copy itself into a data file and format the data file to indicate a template format while maintaining a regular file extension for the data file.

One combination of suspect instructions that is detected by the macro scanning module 304 is a macro enablement instruction and a macro reproduction instruction. A macro enablement instruction is one which formats a file to indicate that the file includes a macro for execution. For example, the file formatting may be set to indicate a template file so that an application program 204 causes the template file to be executed when the file is opened. A macro reproduction instruction is one which allows the macro virus to be replicated. The combination of a macro enablement instruction and a macro reproduction instruction indicates a macro virus since such instructions allow replication and execution of a macro in a destination file, two common characteristics of macro viruses.

In certain application files, such as Microsoft WORD files, if the file format field .format is set to 1, then the application program 204 (WORD) determines that the file may include embedded macros and, at appropriate initiation, will access and execute any macros which are embedded within the file. Thus, setting .format to 1 in a file enables execution of a macro in the file and any macro instruction that seeks to provide such a setting in a destination file may be construed as a macro enablement instruction. For example, the instruction if dlg.format=0 then dlg.format=1 is a macro enablement instruction since it allows the .format to be changed from 0 to 1 in a destination file. Other instructions, such as FileSaveAs a$, 1 keep an original file and save an additional copy of the file under a different format such as one that indicates that the file can include an embedded macro. Therefore, these types of instructions are also macro enablement instructions. Various alternative instructions which enable macro virus execution in files will be recognized.

A macro virus reproduction instruction is a type that allows replication of the macro virus. For example, the MacroCopy instruction copies a macro, and, if the macro is infected, all of its harmful instructions, from a source to a destination. Other instructions, such as Organizer .copy, also facilitate macro virus reproduction. It is understood that various alternative instructions can facilitate macro virus reproduction.

As indicated regarding the preferred method 500 of locating and decoding macros, the macro instructions from the targeted file are located and decoded into binary code for analysis. A unique binary code also corresponds to suspect instructions. For example, the macro virus enablement instruction if dlg.format=0 then dlg.format=1 has a particular corresponding binary code as does the macro virus reproduction instruction MacroCopy. Thus, the comparison data which is obtained 615 from the virus information module 308 may respectively include the binary code for the first and second instructions, or unique portions of the binary codes, to identify the first and second instructions in the macro from a targeted file.

Furthermore, in accordance with the present invention, it has been determined that unique binary code portions may correspond to several suspect instructions. For example, the binary string 73 CB 00 0C 6C 01 00 (in hexadecimal form) corresponds to the instruction portion .format=1 which is found in several macro virus enablement instructions. For example, the instruction if dlg.format=0 then dlg.format=1 noted above, as well as the instructions if bowaardlg.format=0 then bowarrdlg.format=1; FileSaveAs .Format=1; and FileSaveAs .Name=Filename$(), .Format=1 include the binary string 73 CB 00 0C 6C 01 00. Thus, a particular string such as 73 CB 00 0C 6C 01 00 is used by the present invention as an identifier for detecting a plurality of different suspect macro instructions.

The comparison data in the virus information module 308 preferably includes several sets of instruction identifiers. Various combinations of suspect instructions may be detected using the sets of instruction identifiers. Various different macro virus enablement and/or macro virus reproduction instructions may be identified using each set of instruction identifiers. The instruction identifiers are not restricted to macro virus enablement and reproduction instructions. For example, an instruction which causes the computer hard disk to be reformatted without verification and an instruction which changes the system settings to allow such reformatting without user notice could be used as a suspect instruction combination.

Referring now to FIG. 9, a data table including exemplary sets of instruction identifiers which are stored in the virus information module 308 is shown. The exemplary data table

900 includes rows 902 which correspond to several different sets of instruction identifiers. Columns identifying the sets of instruction identifiers 903, their instruction identifier numbers 904 and text and corresponding hexadecimal representations 905 of the binary code for the instruction identifiers are included. Preferably, two instruction identifiers are included in each set of instruction identifiers, but additional instruction identifiers can be included in a set. Additionally, a positive macro virus determination can be made based upon detection of two out of three instruction identifiers or some other subset of identifiers. The data table 900 is exemplary only. The comparison data may be variously stored in the virus information module 308.

Referring again to the flow chart of FIG. 6, once a set of instruction identifiers is obtained by the macro virus scanning module 304 in step 615, the decoded macro is scanned to determine whether it includes a combination of suspect instructions as identified by the instruction identifiers. In step 620, the decoded macro is scanned using a first instruction identifier. For example, the decoded macro may be scanned 620 to determine whether a string 73 CB 00 0C 6C 01 00 which corresponds to the first instruction identifier in the first set of exemplary instruction identifiers 900 is present. The scanning in step 620 may be undertaken by a state machine which scans the decoded macro and determines whether the string is present. In step 625, it is determined whether the first suspect instruction identifier is present in the decoded macro. If it is determined 625 that the no instruction corresponding to the first instruction identifier is present, then in step 645 it is determined that the macro is not infected according to the set of instruction identifiers and the method of macro virus scanning 600 ends.

Therefore a first set of suspect instruction identifiers as shown in FIG. 9 includes the strings 73 CB 00 0C 6C 01 100 and 67 C2 80. A second set of suspect instruction identifiers includes the strings 73 CB 00 0C 6C 01 00 and 64 6F 02 67 DE 00 73 87 02 12 73 7F. A third set of suspect instruction identifiers includes the strings 73 CB 00 0C 6C 01 00 and 6D 61 63 72 6F 73 76 08. A fourth set of suspect instruction identifiers includes the strings 12 6C 01 00 and 64 67 C2 80 6A 0F 47 and a fifth set of suspect instruction identifiers includes the strings 79 7C 66 6F 72 6D 61 74 20 63 6A and 80 05 6A 07 43 4F 4D.

If it is determined in step 625 that the first instruction identifier is present, then the decoded macro is scanned in step 630 to determine whether the second instruction identifier is present. If in step 635 it is determined that the macro includes the second suspect instruction identifier, then in step 640 the decoded macro is flagged as infected by an unknown virus corresponding to the set of instruction identifiers. Information associating the decoded macro to the set of instruction identifiers that resulted in a positive unknown virus detection is stored in the data buffer 312 so that other modules such as the macro treating module 306 can treat the infected macro accordingly.

If it is determined in step 635 that the second identifier is not present, the macro virus scanning module 304 determines in step 645 that, according to the set of instruction identifiers, an unknown virus is not present in the decoded macro and the preferred method 600 of macro virus scanning ends. The determination in step 635 is provided with regard to a single set of instruction identifiers. Other sets of instruction identifiers can be iteratively compared to the decoded macro to otherwise provide a positive determination of an unknown virus. Additionally, the presence of a common first instruction identifier can be determined prior to searching for various alternative second instruction identifiers.

Referring now to the flow chart of FIG. 7 a preferred method 700 for treating infected macros is shown. In step 705, the known virus determination flag is checked to determine whether the macro virus scanning module 304 detected a known virus in the decoded macro. The known virus determination flag is provided to the macro treating module 306 in the data buffer 312, which also associates the decoded macro to the virus information used to detect the known virus. The virus information is used, in step 715, to remove the known virus from the decoded macro. Preferably, the virus is removed from the macro by replacing it with benign instructions (such as no-ops). Since the virus is known, it can be selectively removed such that the legitimate portions of the macro remain. After removal of the virus, in step 735 the treated macro is checked to verify its integrity. If it is determined that the integrity of the treated macro was maintained, then in step 745 the treated macro is flagged as valid by the macro treating module 306. The treated macro is or remains stored in the data buffer 312 as does the information regarding its validity. If it is determined in step 740 that integrity was not maintained, then the treated macro is flagged as invalid in step 750 and the information is similarly stored in the data buffer.

Referring back to step 710, if it is determined by the macro treating module 306 that a known virus is not present in the decoded macro, then the macro is treated to selectively remove an unknown virus. The set of instruction identifiers that was used to detect an unknown virus in the decoded macro is available to the macro treating module 306 in the data buffer 312. An exemplary set includes first and second suspect instruction identifiers. In step 720, the first suspect instruction identifier is used to locate each suspect instruction which corresponds to the identifier. The decoded macros can be scanned to locate such instructions using the techniques described in connection with detecting the instructions used by the macro virus scanning module 304. If the instruction identifiers correspond to fragments of instructions rather than whole instructions, then the macro treating module 306 correlates each detected fragment to a whole instruction. The correlation will depend upon the programming language used by the macro. Conventional techniques can be used for the correlation. In step 725, the additional suspect instruction identifiers are used to locate corresponding additional suspect instructions. The located suspect instructions are then replaced in step 730. Similar to the replacement of known virus strings, preferably the suspect instructions are replaced with benign instructions. The integrity of the macro is verified and the treated macro is flagged according to whether its integrity was maintained, after which the method 700 of treating macros ends.

Referring now to the flow chart of FIG. 8, a preferred method 800 of file correcting in accordance with the present invention is shown. The file correcting module 310 is in communication with the data buffer 312 as well as the various modules 302, 304, 306, 308, 310 and thereby accesses information such as the macro viruses which were detected and the targeted files that the infected macros were found in. In step 805, a targeted file which includes a macro virus is stored in the data buffer 312. Preferably, the targeted file is accessed in its original location and is copied with its infected macro into the data buffer 312. The file correcting module 310 then takes corrective action by replacing the macro in the targeted file with a treated macro or by taking alternative corrective action. In step 810, the macro validity flag is checked to determine the integrity of the treated macro which corresponds to the targeted file. If the treated macro is indicated to be valid, then the file correcting module 310 replaces the infected macro in the targeted file with a treated macro. In step 810 the infected macro is located within the targeted file. This is done by using the information sharing resources (OLE) of the operating system 202. In step 820, the located macro is removed from the targeted file using the information sharing resources and a version of the targeted file without the macro is stored in the data buffer 312. In step 825, the treated macro which was produced by the macro treating module 306 is added to the version of the targeted file without the macro to provide a corrected file. In step 830, the corrected file is used to replace the targeted file in its original location. The corrected file can directly replace the originally targeted file. Alternatively, the targeted file can be deleted or overwritten and the corrected file stored elsewhere.

Referring back to step 810, if the treated macro which corresponds to the targeted file is flagged as invalid, the treated macro is preferably not used to replace the infected macro. Thus, in step 835, alternative corrective action is taken by the file correcting module dependent upon how it is configured by the user. Various alternative corrective steps will be recognized such as notifying the user that the targeted file includes a virus, removing the infected macro from the targeted file without replacement, or deleting the targeted file.

Although the present invention has been described with reference to certain preferred embodiments, those skilled in the art will recognize that various modifications may be provided. For example, although a sequence of accessing, locating, decoding, detecting and correcting has been described with respect to various modules, it is understood that the various processes may be integrated into common modules which perform equivalent functions in detecting an unknown virus in a macro. These and other variations upon and modifications to the described embodiments are provided for by the present invention which is limited only by the following claims.

We claim:

1. In a computer system comprising a processor and a memory, a method for detecting viruses in macros, the method comprising:

obtaining comparison data including information for detecting a virus;

retrieving a macro;

decoding the macro to produce a decoded macro; and

scanning the decoded macro for a virus by comparing the decoded macro to the comparison data;

wherein the comparison data includes a first suspect instruction identifier and a second suspect instruction identifier;

wherein the scanning the decoded macro comprises:

determining whether the decoded macro includes a first portion which corresponds to the first suspect instruction identifier;

determining whether the decoded macro includes a second portion which corresponds to the second suspect instruction identifier;

determining that the decoded macro includes the virus if the decoded macro includes the first and second portions; and

wherein the first suspect instruction identifier identifies a macro virus enablement instruction.

2. The method of claim 1, further comprising:

removing the virus from the macro to produce a treated macro if the scanning the decoded macro indicates that the macro is infected with the virus.

3. The method of claim 1, wherein the retrieving a macro comprises:

accessing a targeted file;

determining whether the targeted file is a template file;

if the targeted file is not a template file, determining whether the targeted file includes an embedded macro; and

if the targeted file includes an embedded macro, locating the embedded macro.

4. The method of claim 1, wherein the second suspect instruction identifier detects a macro virus reproduction instruction.

5. The method of claim 1, wherein the removing the virus comprises:

locating a first suspect macro instruction in the decoded macro which corresponds to the first suspect instruction identifier; and

removing the first suspect macro instruction.

6. The method of claim 5, wherein the removing the first suspect macro instruction includes replacing the first suspect instruction with a benign instruction.

7. The method of claim 5, wherein the removing the virus comprises:

locating a second suspect macro instruction in the decoded macro which corresponds to the second suspect instruction identifier; and

removing the second suspect macro instruction from the decoded macro to produce a treated macro.

8. The method of claim 1, wherein the comparison data includes a plurality of sets of suspect instruction identifiers.

9. In a computer system comprising a processor and a memory, a method for detecting viruses in macros, the method comprising:

obtaining comparison data including information for detecting a virus;

retrieving a macro;

decoding the macro to produce a decoded macro;

scanning the decoded macro for a virus by comparing the decoded macro to the comparison data; and

removing the virus from the macro to produce a treated macro if the step of scanning the decoded macro indicates that the macro is infected with the virus;

verifying the integrity of the treated macro; and

replacing the infected macro in a targeted file with the treated macro dependent upon the integrity verification of the treated macro.

10. In a computer system comprising a processor and a memory, a method for detecting viruses in macros, the method comprising:

obtaining comparison data including information for detecting a virus;

retrieving a macro;

decoding the macro to produce a decoded macro;

scanning the decoded macro for a virus by comparing the decoded macro to the comparison data;

wherein the comparison data includes a first suspect instruction identifier and a second suspect instruction identifiers; and

wherein a first set of respective first and second suspect instruction identifiers comprises the strings 73 CB 00 0C 6C 01 00 and 67 C2 80.

11. The method of claim 10, wherein a second set of respective first and second suspect instruction identifiers

comprises the strings 73 CB 00 0C 6C 01 00 and 64 6F 02 67 DE 00 73 87 01 12 73 7F, a third set of respective first and second suspect instruction identifiers comprises the strings 73 CB 00 0C 6C 01 00 and 6D 61 63 72 6F 73 76 08, a fourth set of respective first and second suspect instruction identifiers comprises the strings 12 6C 01 00 and 64 67 C2 80 6A 0F 47, and a fifth set of respective first and second suspect instruction identifiers comprises the strings 79 7C 66 6F 72 6D 61 74 20 63 6A and 80 05 6A 07 43 4F 4D.

12. In a computer system comprising a processor and a memory, a method for detecting viruses in macros, the method comprising:

retrieving a macro;

obtaining comparison data for detecting a virus, the comparison data including a first suspect instruction identifier and a second suspect instruction identifier;

scanning the macro to determine whether the macro includes a first portion which corresponds to the first suspect instruction identifier;

scanning the macro to determine whether the macro includes a second portion which corresponds to the second suspect instruction identifier; and

determining that the macro is infected with the virus if the macro includes the first and second portions;

wherein the first suspect instruction identifier includes the string 73 CB 00 0C 6C 01 00 and the second suspect instruction identifier includes the string 67 C2 80

13. The method of claim 12, further comprising:

treating the macro to produce a treated macro if it is determined that the macro includes the first and second portions.

14. The method of claim 13, wherein the treating the macro comprises:

locating a first macro instruction in the infected macro which corresponds to the first suspect instruction identifier; and

removing the first macro instruction from the infected macro to repair the infected macro.

15. The method of claim 14, wherein the treating the macro comprises:

locating a second macro instruction in the infected macro which corresponds to the second suspect instruction identifier; and

removing the second macro instruction from the infected macro to repair the infected macro.

16. The method of claim 12, wherein the retrieving a macro comprises:

accessing a targeted file; and

determining whether the targeted file is a template file;

if the file is not a template file, determining whether the targeted file includes an embedded macro; and

if the file includes an embedded macro, locating the embedded macro.

17. The method of claim 12, wherein the comparison data includes a plurality of sets of suspect instruction identifiers.

18. In a computer system comprising a processor and a memory, a method for detecting viruses in macros, the method comprising:

retrieving a macro;

obtaining comparison data for detecting a virus, the comparison data including a first suspect instruction identifier and a second suspect instruction identifier;

scanning the macro to determine whether the macro includes a first portion which corresponds to the first suspect instruction identifier;

scanning the macro to determine whether the macro includes a second portion which corresponds to the second suspect instruction identifier;

determining that the macro is infected with the virus if the macro includes the first and second portions,

wherein the comparison data includes a plurality of sets of respective first and second suspect instruction identifiers; and

wherein a first set of suspect instruction identifiers comprises the strings 73 CB 00 0C 6C 01 00 and 67 C2 80, a second set of suspect instruction comprises the strings 73 CB 00 0C 6C 01 00 and 64 6F 02 67 DE 00 73 87 01 12 73 7F, a third set of suspect instruction identifiers comprises the strings 73 CB 00 0C 6C 01 00 and 6D 61 63 72 6F 73 76 08, a fourth set of suspect instruction identifiers comprises the strings 12 6C 01 00 and 64 67 C2 80 6A 0F 47, and a fifth set of suspect instruction identifiers comprises the strings 79 7C 66 6F 72 6D 61 74 20 63 6A and 80 05 6A 07 43 4F 4D.

19. In a computer system comprising a processor and a memory, a method for detecting viruses in macros, the method comprising:

retrieving a macro;

obtaining comparison data for detecting a virus, the comparison data including a first suspect instruction identifier and a second suspect instruction identifier;

scanning the macro to determine whether the macro includes a first portion which corresponds to the first suspect instruction identifier;

scanning the macro to determine whether the macro includes a second portion which corresponds to the second suspect instruction identifier;

determining that the macro is infected with the virus if the macro includes the first and second portions; and

treating the macro to produce a treated macro if it is determined that the macro includes the firs and second portions, further comprising:

accessing a targeted file;

locating a macro within the targeted file;

removing the macro from the targeted file; and

adding the treated macro to the targeted file to produce a corrected file.

20. An apparatus for detecting viruses in macros, the apparatus comprising:

a virus information module, for storing comparison data for detecting a virus, the comparison data including a first suspect instruction identifier and a second suspect instruction identifier;

a macro virus scanning module, in communication with the virus information module, for receiving the comparison data and scanning a macro to determine whether the macro includes a first portion which corresponds to the first suspect instruction identifier and a second portion which corresponds to the second suspect instruction identifier;

a macro locating and decoding module, in communication with the macro virus scanning module, for accessing a targeted file, determining whether the targeted file is a template file, determining whether the targeted file includes an embedded macro, and decoding the macro to produce a decoded macro;

a macro treating module, in communication with the virus information module, for accessing the decoded macro and removing a first macro instruction which corresponds to the first suspect instruction identifier and a second macro instruction which corresponds to the second suspect instruction identifier to produce a treated macro; and

a file correcting module, in communication with the macro treating module, for accessing the targeted file, locating the macro within the targeted file, removing the macro from the targeted file and adding the treated macro to the targeted file to produce a corrected file.

21. The apparatus of claim 20, wherein the comparison data includes a plurality of sets of suspect instruction identifiers.

22. An apparatus for detecting viruses in macros, the apparatus comprising:

a virus information module, for storing comparison data for detecting a virus, the comparison data including a first suspect instruction identifier and a second suspect instruction identifier; and

a macro virus scanning module, in communication with the virus information module, for receiving the comparison data and scanning a macro to determine whether the macro includes a first portion which corresponds to the first suspect instruction identifier and a second portion which corresponds to the second suspect instruction identifier, wherein the first instruction identifier includes the string 73 CB 00 0C 6C 01 00 and the second suspect instruction identifier includes the string 67 C2 80.

23. An apparatus for detecting viruses in macros, the apparatus comprising:

a virus information module, for storing comparison data for detecting a virus, the comparison data including a first suspect instruction identifier and a second suspect instruction identifier; and

a macro virus scanning module, in communication with the virus information module, for receiving the comparison data and scanning a macro to determine whether the macro includes a first portion which corresponds to the first suspect instruction identifier and a second portion which corresponds to the second suspect instruction identifier;

wherein the comparison data includes a plurality of sets of respective first and second suspect instruction identifiers; and

wherein a first set of suspect instruction identifiers comprises the strings 73 CB 00 0C 6C 01 00 and 67 C2 80, a second set of suspect instruction comprises the strings 73 CB 00 0C 6C 01 00 and 64 6F 02 67 DE 00 73 87 01 12 73 7F, a third set of suspect instruction identifiers comprises the strings 73 CB 00 0C 6C 01 00 and 6D 61 63 72 6F 73 76 08, a fourth set of suspect instruction identifiers comprises the strings 12 6C 01 00 and 64 67 C2 80 6A 0F 47, and a fifth set of suspect instruction identifiers comprises the strings 79 7C 66 6F 72 6D 61 74 20 63 6A and 80 05 6A 07 43 4F 4D.

24. An apparatus for detecting viruses in macros, the apparatus comprising:

means for obtaining comparison data for detecting a virus, the comparison data including a first suspect instruction identifier and a second suspect instruction identifier;

means for scanning the macro to determine whether a macro includes a first portion which corresponds to the first suspect instruction identifier;

means for scanning the macro to determine whether the macro includes a second portion which corresponds to the second suspect instruction identifier;

means for determining that the macro is infected with the virus if the macro includes the first and second portions;

means for accessing a targeted file and determining whether the targeted file includes a macro; and

means for correcting a file, the means for correcting a file including means for accessing the targeted file, means for removing the macro from the targeted file and means for adding the treated macro to the targeted file to produce a corrected file.

25. The apparatus of claim 24, further comprising:

means for locating a first macro instruction and a second macro instruction within the macro which respectively correspond to the first suspect instruction identifier and the second suspect instruction identifier; and

means for removing the first macro instruction and the second macro instruction from the macro to produce a treated macro.

* * * * *

EXHIBIT 9

US005740441A

United States Patent [19]

Yellin et al.

[11] **Patent Number: 5,740,441**

[45] **Date of Patent: Apr. 14, 1998**

[54] **BYTECODE PROGRAM INTERPRETER APPARATUS AND METHOD WITH PRE-VERIFICATION OF DATA TYPE RESTRICTIONS AND OBJECT INITIALIZATION**

[75] Inventors: **Frank Yellin**, Redwood City; **James A. Gosling**, Woodside, both of Calif.

[73] Assignee: **Sun Microsystems, Inc.**, Palo Alto, Calif.

[21] Appl. No.: **575,291**

[22] Filed: **Dec. 20, 1995**

Related U.S. Application Data

[63] Continuation-in-part of Ser. No. 360,202, Dec. 20, 1994.

[51] **Int. Cl.**^{6} **G06F 9/45**

[52] **U.S. Cl.** 395/704; 395/705; 395/707

[58] **Field of Search** 395/704, 705, 395/707

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,179,734	1/1993	Candy et al.	395/800
5,450,575	9/1995	Sites	395/700
5,590,329	12/1996	Goodnow, II et al.	395/708

OTHER PUBLICATIONS

Ken Thompson, "Regular Expression Search Algorithm," *Communications of the ACM*, Jun. 1968, vol. 11, No. 6, pp. 419–422.

Kin–Man Chung and Herbert Yuen, "A 'Tiny' Pascal Compiler; Part 1: The P–Code Interpreter," *BYTE Publications, Inc.*, Sep. 1978.

Kin–Man Chung and Herbert Yuen, "A 'Tiny' Pascal Compiler; Part 2: The P–Compiler," *BYTE Publications, Inc.*, Oct. 1978.

Gene McDaniel, "An Analysis of a Mesa Instruction Set," *Association for Computing Machinery*, May 1982.

Kenneth A. Pier, "A Retrospective on the Dorado, A High–Performance Personal Computer," *IEEE Computer Society; 10th Annual Intl. Symposium on Computer Architecture*, 1983, pp. 252–269.

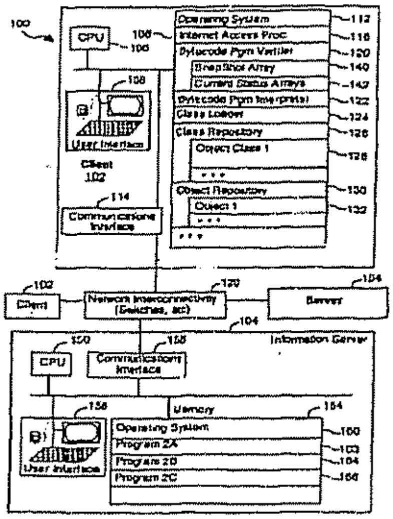
James G. Mitchell, et al., "Mesa Language Manual," *Xerox Corporation, Palo Alto Research Center*.

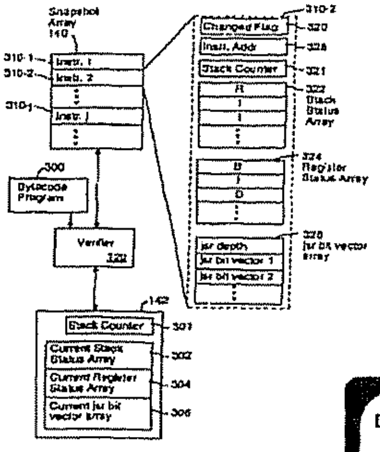
Primary Examiner—Lucien U. Toplu
Attorney, Agent, or Firm—Gary S. Williams; Flehr Hohbach Test Albritton & Herbert LLP

[57] **ABSTRACT**

A program interpreter for computer programs written in a bytecode language, which uses a restricted set of data type specific bytecodes. The interpreter, prior to executing any bytecode program, executes a bytecode program verifier procedure that verifies the integrity of a specified program by identifying any bytecode instruction that would process data of the wrong type for such a bytecode and any bytecode instruction sequences in the specified program that would cause underflow or overflow of the operand stack. If the program verifier finds any instructions that violate predefined stack usage and data type usage restrictions, execution of the program by the interpreter is prevented. After pre-processing of the program by the verifier, if no program faults were found, the interpreter executes the program without performing operand stack overflow and underflow checks and without performing data type checks on operands stored in operand stack. As a result, program execution speed is greatly improved.

18 Claims, 11 Drawing Sheets





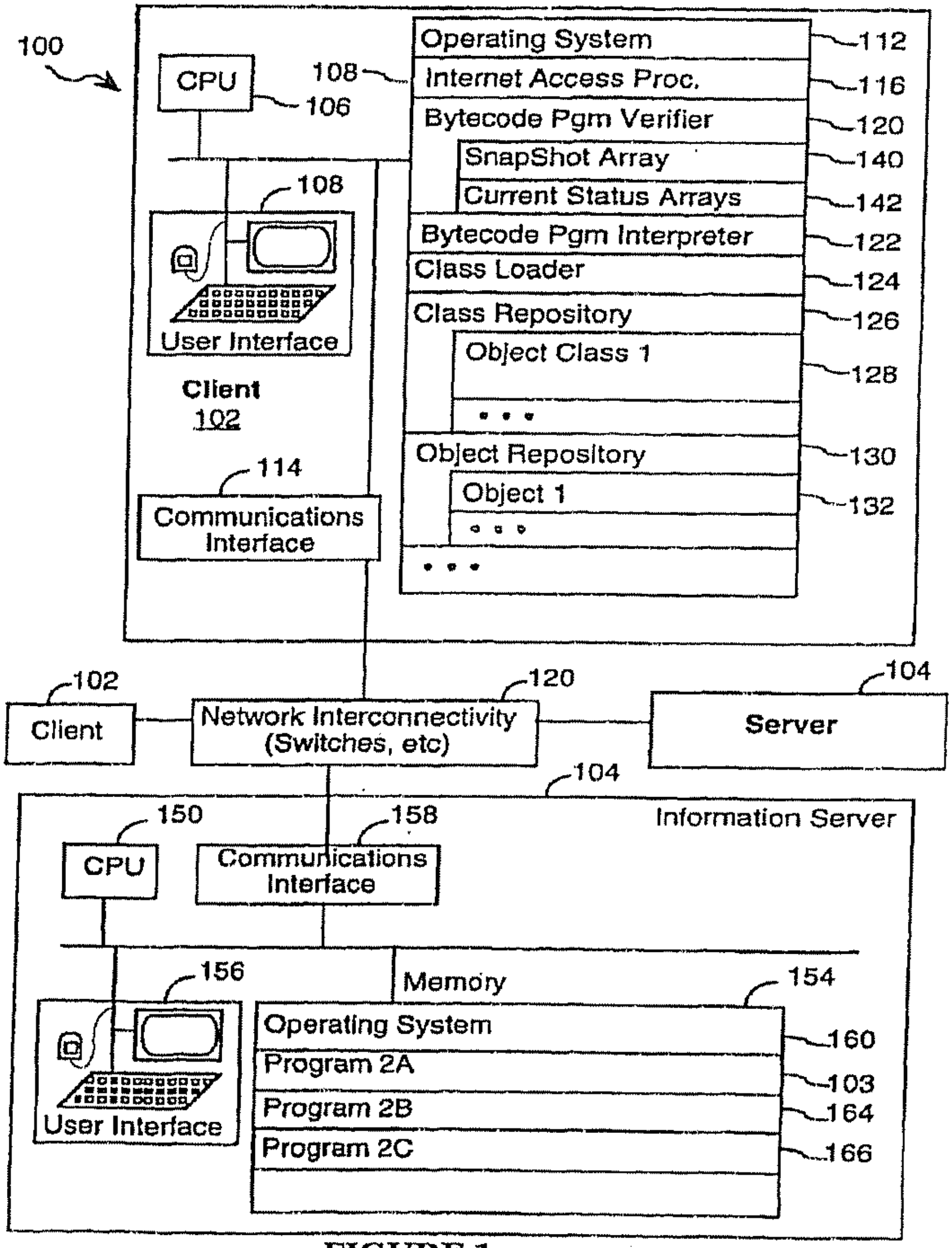

100
CPU
108
106
108
User Interface
Client
102
114
Communications Interface
Operating System
Internet Access Proc.
Bytecode Pgm Verifier
SnapShot Array
Current Status Arrays
Bytecode Pgm Interpreter
Class Loader
Class Repository
Object Class 1
• • •
Object Repository
Object 1
• • •
• • •
112
116
120
140
142
122
124
126
128
130
132
102
Client
120
Network Interconnectivity (Switches, etc)
104
Server
104
Information Server
150
CPU
158
Communications Interface
156
User Interface
Memory
154
Operating System
Program 2A
Program 2B
Program 2C
160
103
164
166

FIGURE 1

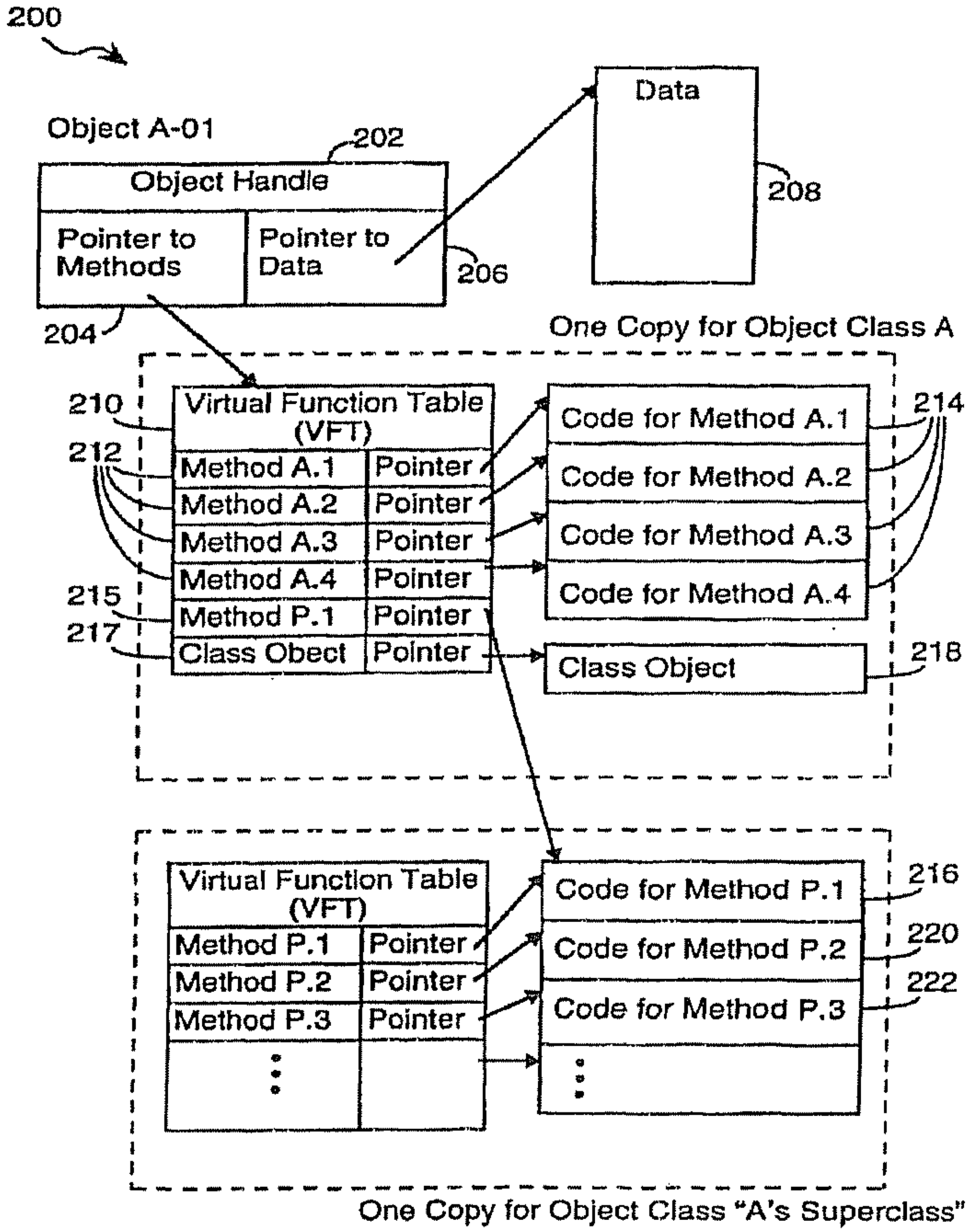

200
Object A-01
202
Object Handle
Pointer to Methods
Pointer to Data
206
204
Data
208
One Copy for Object Class A
210
Virtual Function Table (VFT)
212
Method A.1
Pointer
Method A.2
Pointer
Method A.3
Pointer
Method A.4
Pointer
215
Method P.1
Pointer
217
Class Obect
Pointer
214
Code for Method A.1
Code for Method A.2
Code for Method A.3
Code for Method A.4
Class Object
218
Virtual Function Table (VFT)
Method P.1
Pointer
Method P.2
Pointer
Method P.3
Pointer
Code for Method P.1
216
Code for Method P.2
220
Code for Method P.3
222
One Copy for Object Class "A's Superclass"

FIGURE 2

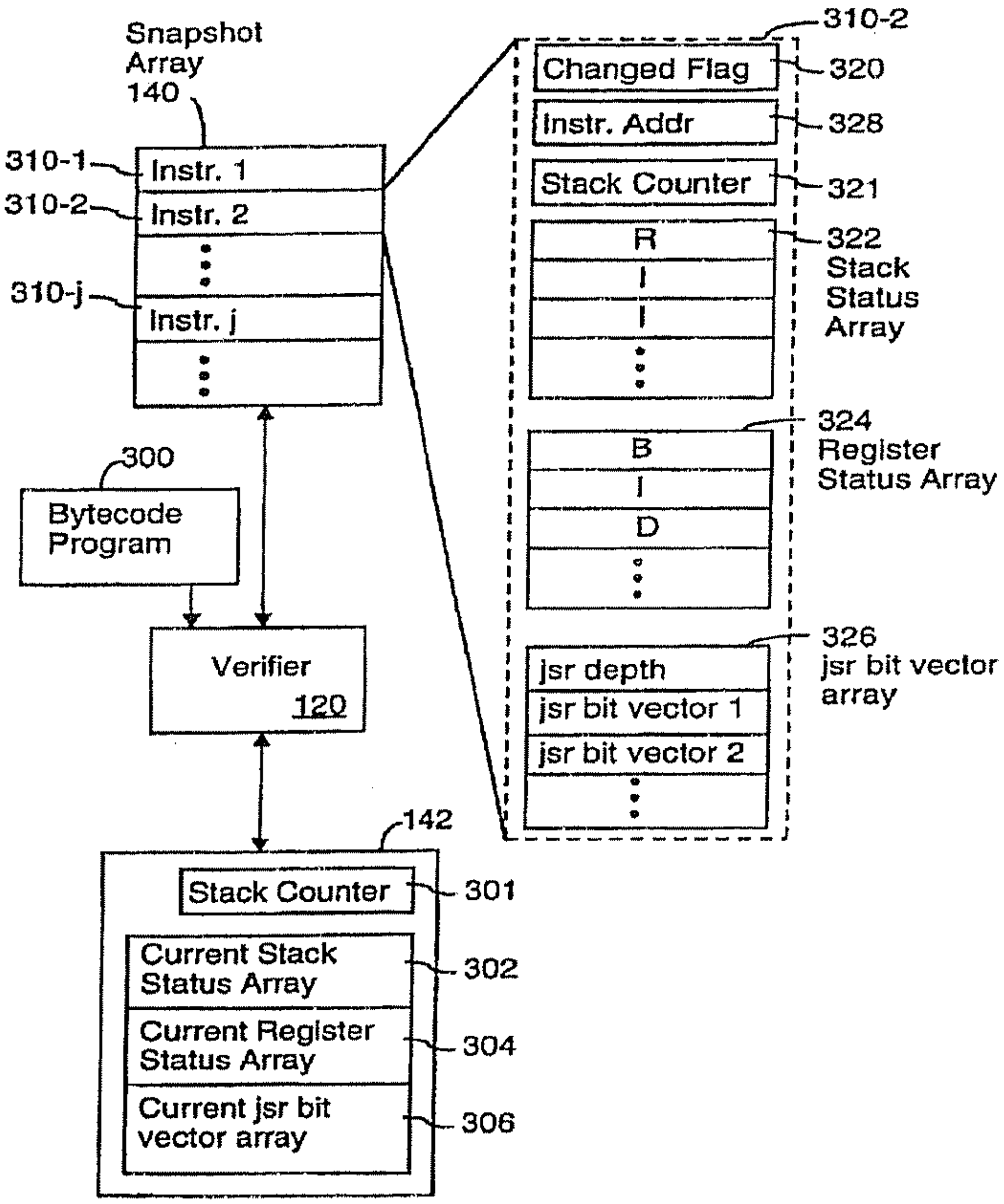

Snapshot Array 140
310-1
Instr. 1
310-2
Instr. 2
310-j
Instr. j
300
Bytecode Program
Verifier 120
142
Stack Counter
301
Current Stack Status Array
302
Current Register Status Array
304
Current jsr bit vector array
306
310-2
Changed Flag
320
Instr. Addr
328
Stack Counter
321
R
I
I
322 Stack Status Array
B
I
D
324 Register Status Array
jsr depth
jsr bit vector 1
jsr bit vector 2
326 jsr bit vector array

FIGURE 3

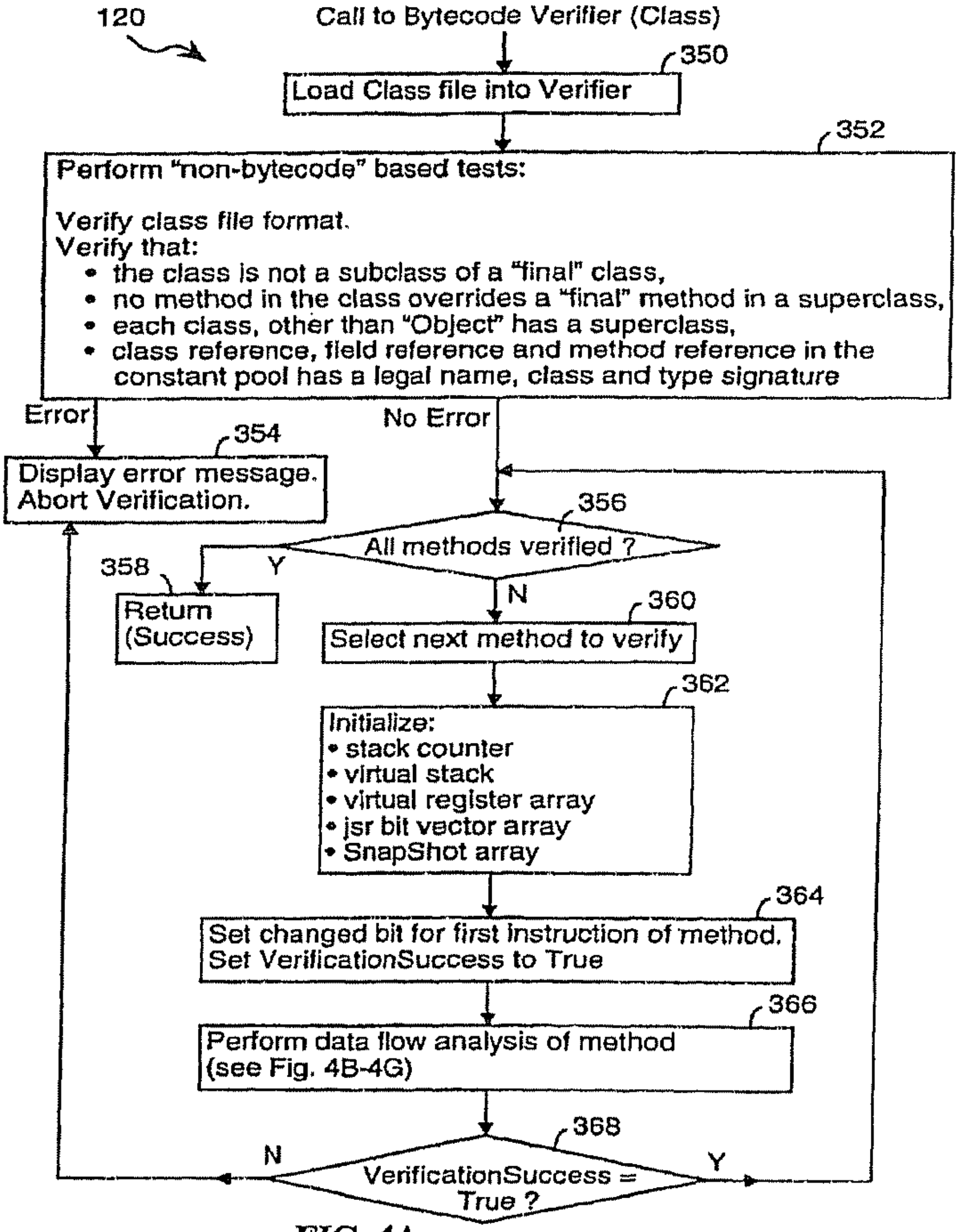

120
Call to Bytecode Verifier (Class)
350
Load Class file into Verifier
352
Perform "non-bytecode" based tests:
Verify class file format.
Verify that:
• the class is not a subclass of a "final" class,
• no method in the class overrides a "final" method in a superclass,
• each class, other than "Object" has a superclass,
• class reference, field reference and method reference in the constant pool has a legal name, class and type signature
Error
No Error
354
Display error message. Abort Verification.
356
All methods verified ?
358
Y
Return (Success)
N
360
Select next method to verify
362
Initialize:
• stack counter
• virtual stack
• virtual register array
• jsr bit vector array
• SnapShot array
364
Set changed bit for first instruction of method. Set VerificationSuccess to True
366
Perform data flow analysis of method (see Fig. 4B-4G)
368
N
VerificationSuccess = True ?
Y

FIG. 4A

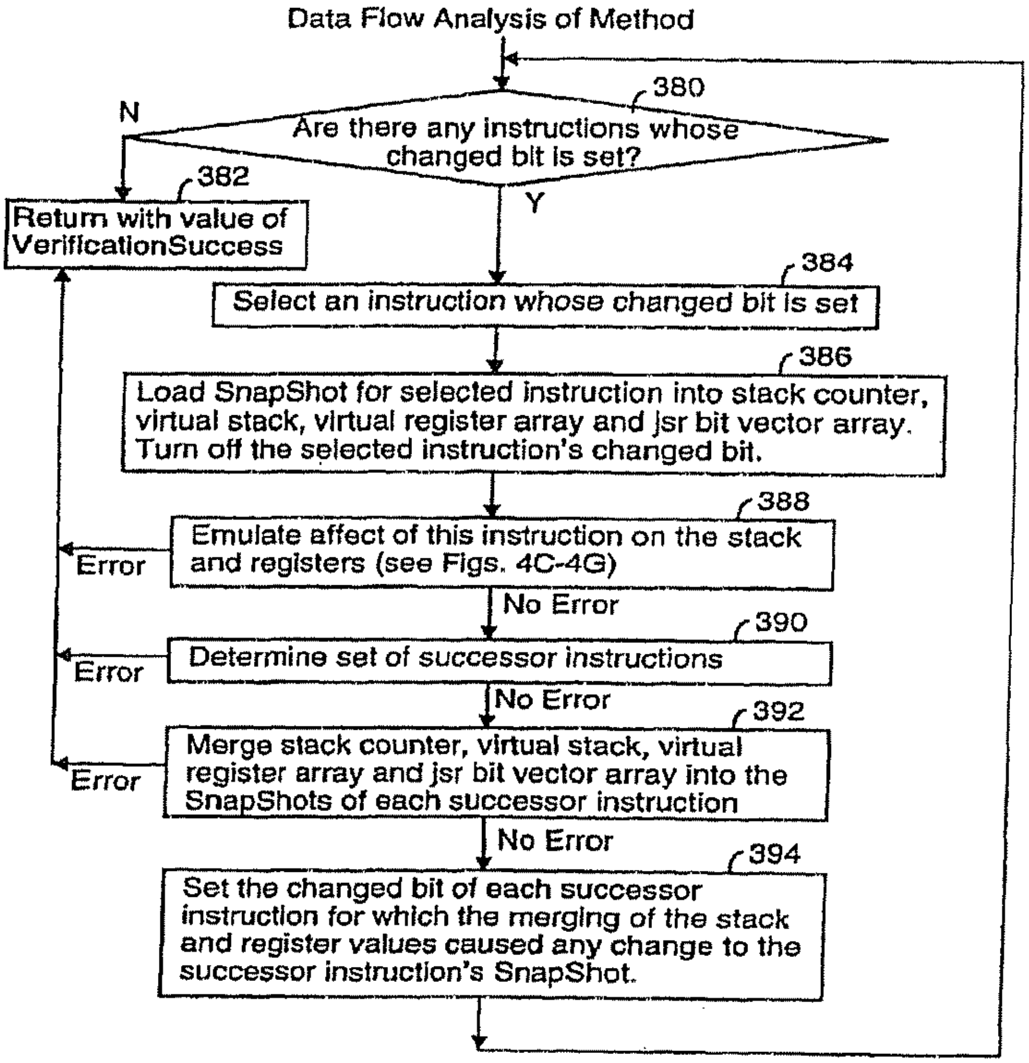

Data Flow Analysis of Method
380
Are there any instructions whose changed bit is set?
N
Y
382
Return with value of VerificationSuccess
384
Select an instruction whose changed bit is set
386
Load SnapShot for selected instruction into stack counter, virtual stack, virtual register array and jsr bit vector array. Turn off the selected instruction's changed bit.
388
Emulate affect of this instruction on the stack and registers (see Figs. 4C-4G)
Error
No Error
390
Determine set of successor instructions
Error
No Error
392
Merge stack counter, virtual stack, virtual register array and jsr bit vector array into the SnapShots of each successor instruction
Error
No Error
394
Set the changed bit of each successor instruction for which the merging of the stack and register values caused any change to the successor instruction's SnapShot.

FIG. 4B

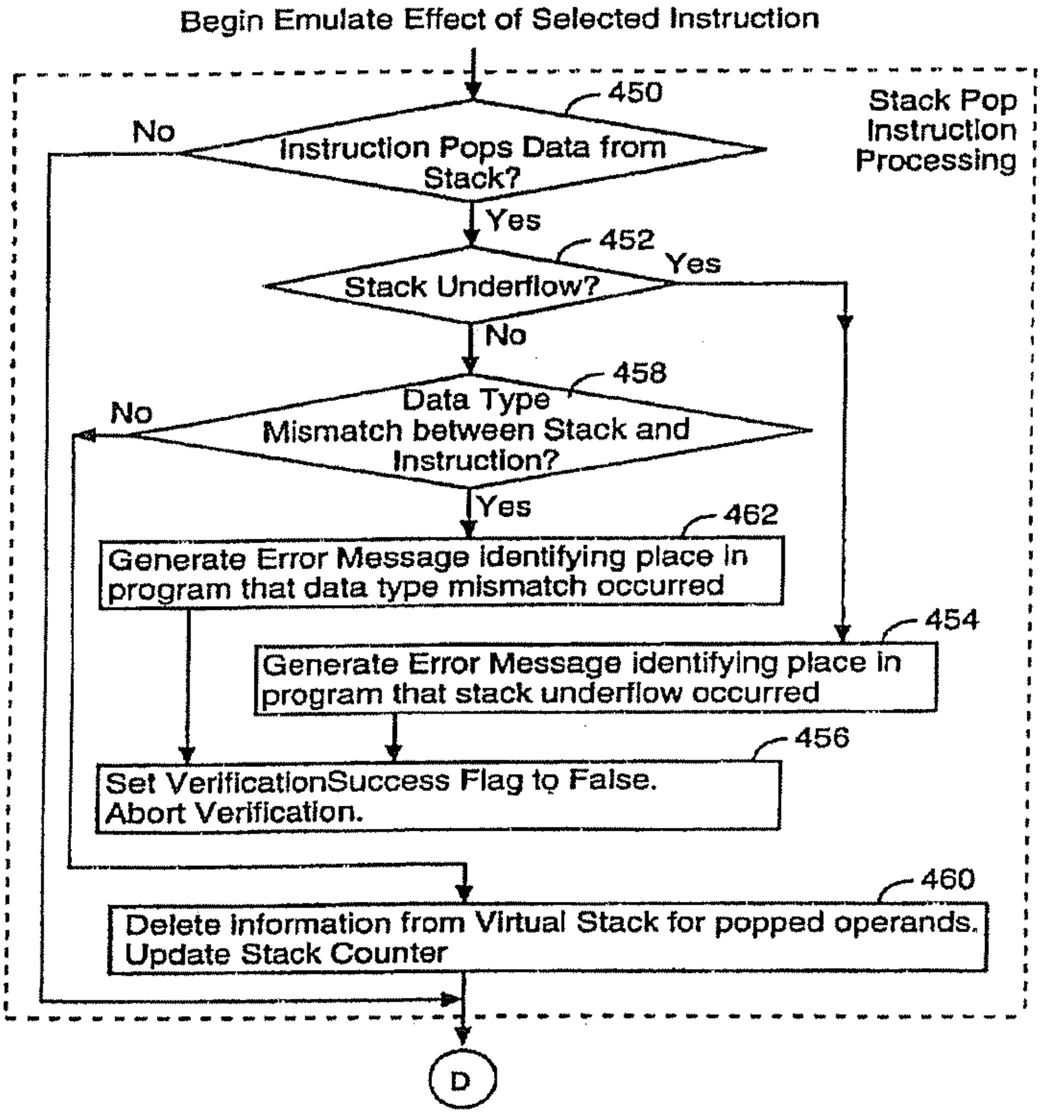

Begin Emulate Effect of Selected Instruction
Stack Pop Instruction Processing
450
Instruction Pops Data from Stack?
No
Yes
452
Stack Underflow?
Yes
No
458
Data Type Mismatch between Stack and Instruction?
No
Yes
462
Generate Error Message identifying place in program that data type mismatch occurred
454
Generate Error Message identifying place in program that stack underflow occurred
456
Set VerificationSuccess Flag to False. Abort Verification.
460
Delete Information from Virtual Stack for popped operands. Update Stack Counter
D

FIG. 4C

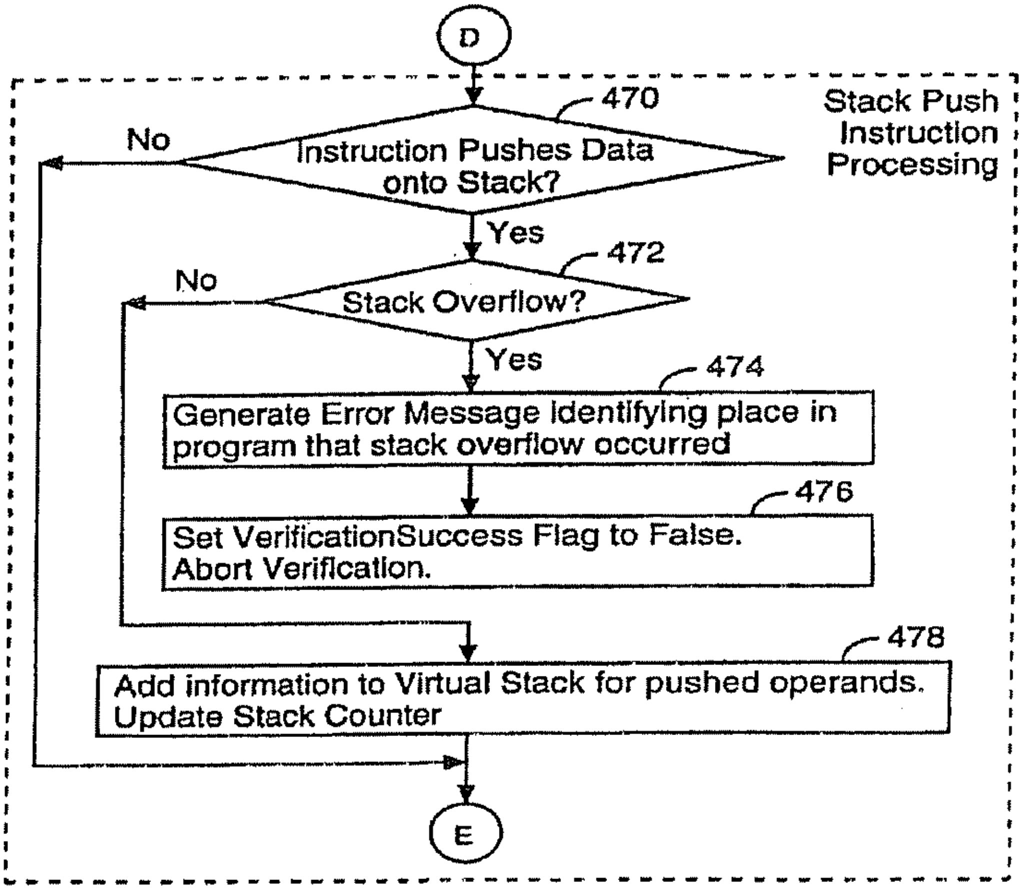

D
470
Stack Push Instruction Processing
No
Instruction Pushes Data onto Stack?
Yes
472
No
Stack Overflow?
Yes
474
Generate Error Message Identifying place in program that stack overflow occurred
476
Set VerificationSuccess Flag to False. Abort Verification.
478
Add information to Virtual Stack for pushed operands. Update Stack Counter
E

FIG. 4D

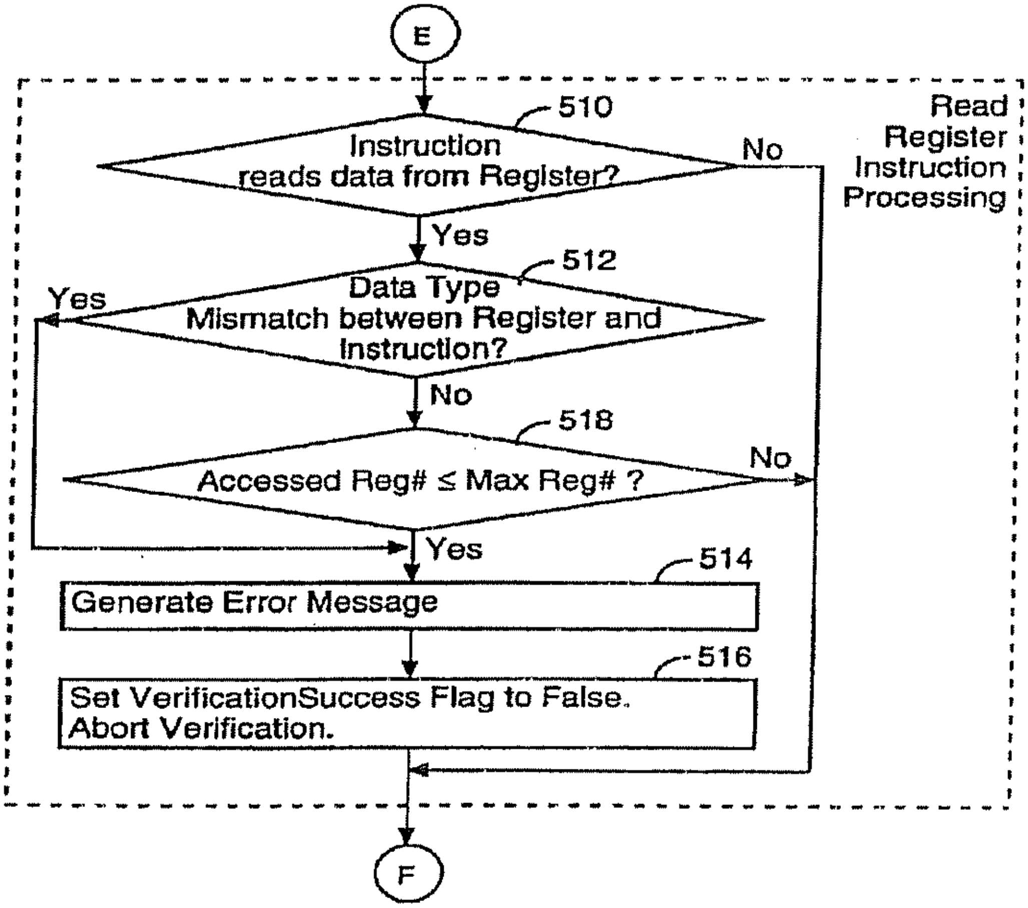

E
510
Instruction reads data from Register?
No
Yes
Read Register Instruction Processing
512
Data Type Mismatch between Register and Instruction?
Yes
No
518
Accessed Reg# ≤ Max Reg# ?
No
Yes
514
Generate Error Message
516
Set VerificationSuccess Flag to False. Abort Verification.
F

FIG. 4E

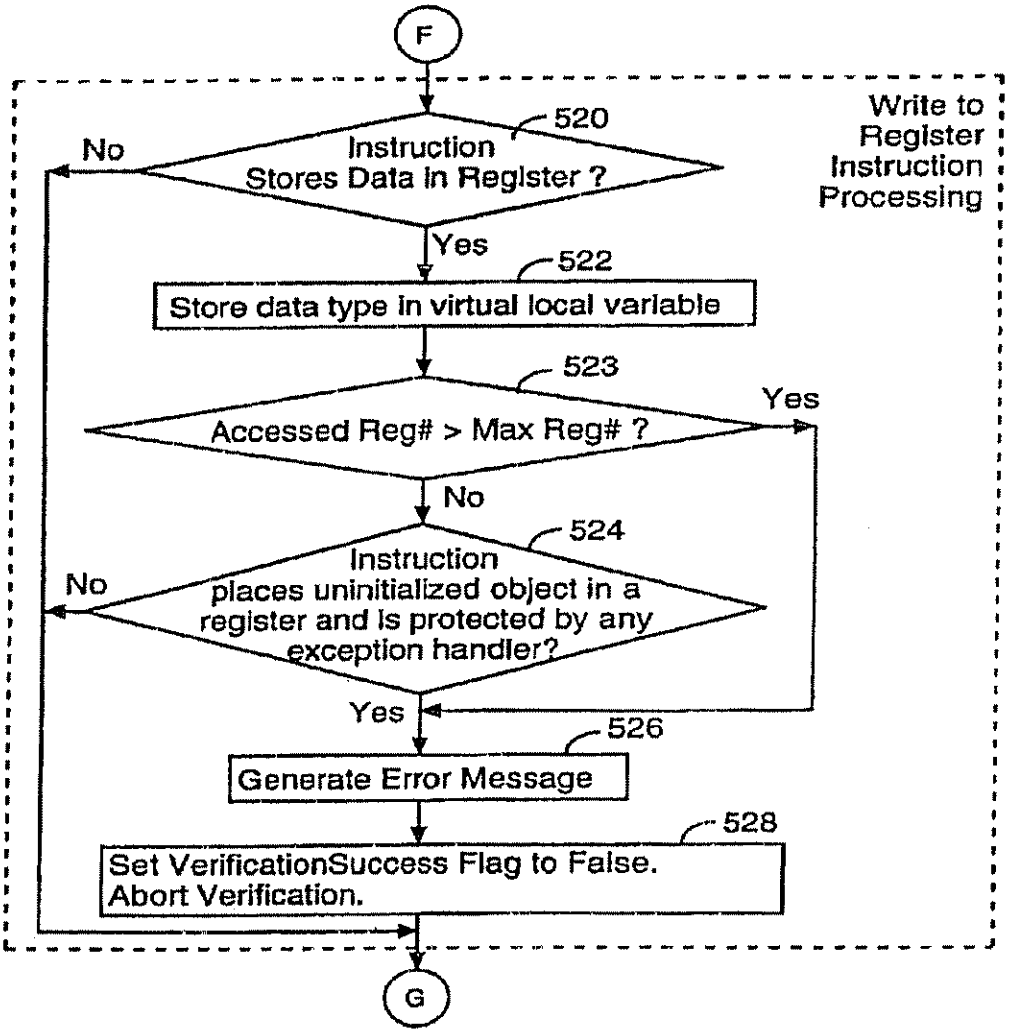

F
Write to Register Instruction Processing
520
Instruction Stores Data in Register ?
No
Yes
522
Store data type in virtual local variable
523
Accessed Reg# > Max Reg# ?
Yes
No
524
Instruction places uninitialized object in a register and is protected by any exception handler?
No
Yes
526
Generate Error Message
528
Set VerificationSuccess Flag to False.
Abort Verification.
G

FIG. 4F

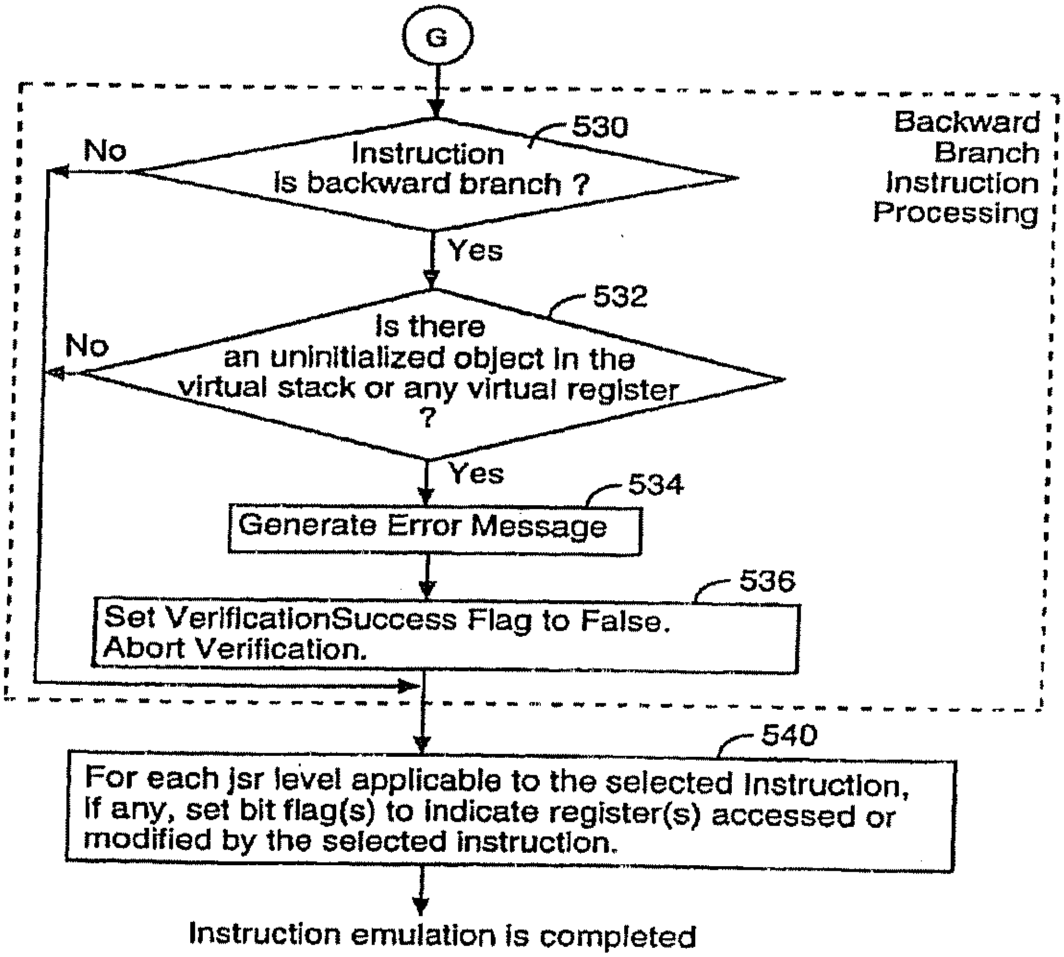

G
Backward Branch Instruction Processing
530
Instruction is backward branch ?
No
Yes
532
Is there an uninitialized object in the virtual stack or any virtual register ?
No
Yes
534
Generate Error Message
536
Set VerificationSuccess Flag to False. Abort Verification.
540
For each jsr level applicable to the selected instruction, if any, set bit flag(s) to indicate register(s) accessed or modified by the selected instruction.
Instruction emulation is completed

FIG. 4G

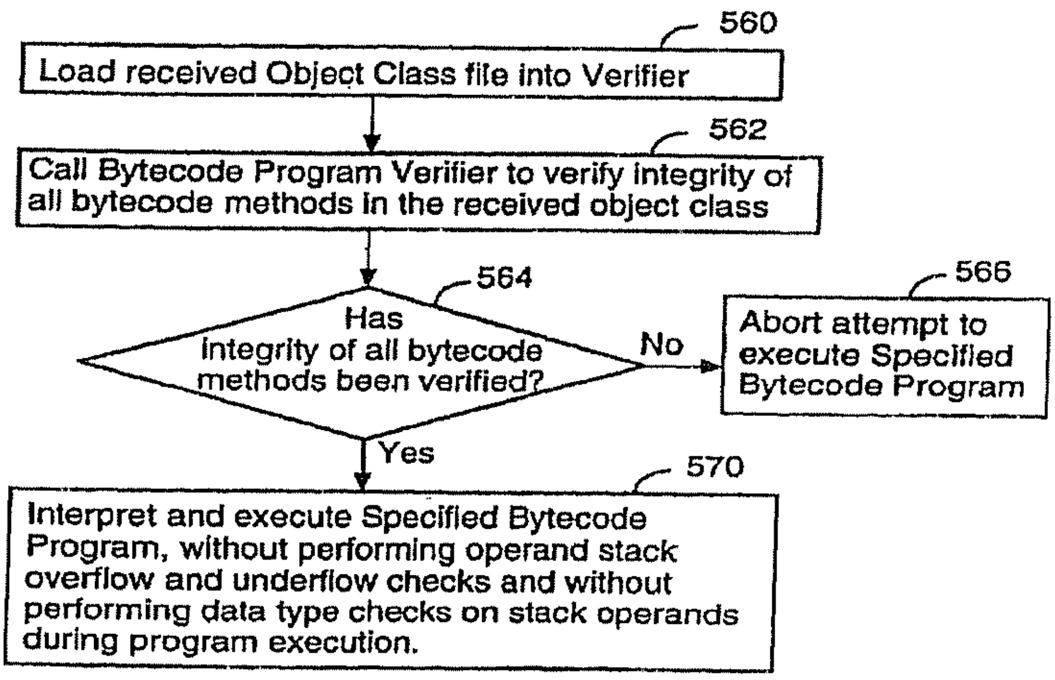

560
Load received Object Class file into Verifier
562
Call Bytecode Program Verifier to verify integrity of all bytecode methods in the received object class
564
Has integrity of all bytecode methods been verified?
No
566
Abort attempt to execute Specified Bytecode Program
Yes
570
Interpret and execute Specified Bytecode Program, without performing operand stack overflow and underflow checks and without performing data type checks on stack operands during program execution.

FIGURE 5

BYTECODE PROGRAM INTERPRETER APPARATUS AND METHOD WITH PRE-VERIFICATION OF DATA TYPE RESTRICTIONS AND OBJECT INITIALIZATION

This application is a continuation-in-part of U.S. application Ser. No. 08/360,202, filed Dec. 20, 1994.

The present invention relates generally to the use of computer software on multiple computer platforms which use distinct underlying machine instruction sets, and more specifically to an program verifier and method that verify the integrity of computer software obtained from a network server or other source.

BACKGROUND OF THE INVENTION

Referring to FIG. 1, in a networked computer system 100, a first computer 102 may download a computer program 103 residing on a second computer 104. In this example, the first user node 102 will typically be a user workstation (often called a client) having a central processing unit 106, a user interface 108, memory 110 (e.g., random access memory and disk memory) for storing an operating system 112, programs, documents and other data, and a communications interface 114 for connecting to a computer network 120 such as the Internet, a local area network or a wide area network. The computers 102 and 104 are often called "nodes on the network" or "network nodes."

The second computer 104 will often be a network server, but may be a second user workstation, and typically would contain the same basic array of computer components as the first computer.

In the prior art (unlike the system shown in FIG. 1), after the first computer 102 downloads a copy of a computer program 103 from the second computer 104, there are essentially no standardized tools available to help the user of the first computer 102 to verify the integrity of the downloaded program 103. In particular, unless the first computer user studies the source code of the downloaded program, it is virtually impossible using prior art tools to determine whether the downloaded program 103 will underflow or overflow its stack, or whether the downloaded program 103 will violate files and other resources on the user's computer.

A second issue with regard to downloading computer software from one computer to another concerns transferring computer software between computer platforms which use distinct underlying machine instruction sets. There are some prior art examples of platform independent computer programs and platform independent computer programming languages. What the prior art lacks are reliable and automated software verification tools for enabling recipients of such software to verify the integrity of transferred platform independent computer software obtained from a network server or other source.

SUMMARY OF THE INVENTION

The present invention verifies the integrity of computer programs written in a bytecode language, commercialized as the JAVA bytecode language, which uses a restricted set of data type specific bytecodes. All the available source code bytecodes in the language either (A) are stack data consuming bytecodes that have associated data type restrictions as to the types of data that can be processed by each such bytecode, (B) do not utilize stack data but affect the stack by either adding data of known data type to the stack or by removing data from the stack without regard to data type, or (C) neither use stack data nor add data to the stack.

The present invention provides a verifier tool and method for identifying, prior to execution of a bytecode program, any instruction sequence that attempts to process data of the wrong type for such a bytecode or if the execution of any bytecode instructions in the specified program would cause underflow or overflow of the operand stack, and to prevent the use of such a program.

The bytecode program verifier of the present invention includes a virtual operand stack for temporarily storing stack information indicative of data stored in a program operand stack during the actual execution a specified bytecode program. The verifier processes the specified program using data flow analysis, processing each bytecode instruction of the program whose stack and register input status map is affected by another instruction processed by the verifier. A stack and register input status map is generated for every analyzed bytecode instruction, and when an instruction is a successor to multiple other instructions, its status map is generated by merging the status maps created during the processing of each of the predecessor instructions. The verifier also compares the stack and register status map information with data type restrictions associated with each bytecode instruction so as to determine if the operand stack or registers during program execution would contain data inconsistent with the data type restrictions of the bytecode instruction, and also determines if any bytecode instructions in the specified program would cause underflow or overflow of the operand stack.

The merger of stack and register status maps requires special handling for the instructions associated with exception handlers and the instructions associated with subroutine calls (including "finally" instruction blocks that are executed via a subroutine call whenever a protected code block is exited).

After pre-processing of the program by the verifier, if no program faults were found, a bytecode program interpreter executes the program without performing operand stack overflow and underflow checks and without performing data type checks on operands stored in operand stack. As a result, program execution speed is greatly improved.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated in and form a part of this specification, illustrate embodiments of the invention and, together with the description, serve to explain the principles of the invention, wherein:

FIG. 1 is a block diagram of a computer system incorporating a preferred embodiment of the present invention

FIG. 2 is a block diagram of the data structure for an object in a preferred embodiment of the present invention.

FIG. 3 is a block diagram of the data structures maintained by a bytecode verifier during verification of a bytecode program in accordance with the present invention.

FIGS. 4A–4G represents flow charts of the bytecode program verification process in the preferred embodiment of the present invention.

FIG. 5 represents a flow chart of the class loader and bytecode program interpreter process in the preferred embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Reference will now be made in detail to the preferred embodiments of the invention, examples of which are illustrated in the accompanying drawings. While the invention will be described in conjunction with the preferred embodiments, it will be understood that they are not intended to limit the invention to those embodiments. On the contrary, the invention is intended to cover alternatives, modifications and equivalents, which may be included within the spirit and scope of the invention as defined by the appended claims.

Referring now to a distributed computer system 100 as shown in FIG. 1, there is shown a distributed computer system 100 having multiple client computers 102 and multiple server computers 104. In the preferred embodiment, each client computer 102 is connected to the servers 104 via the Internet 120, although other types of communication connections could be used. While most client computers are desktop computers, such as Sun workstations, IBM compatible computers and Macintosh computers, virtually any type of computer can be a client computer. In the preferred embodiment, each client computer includes a CPU 106, a user interface 108, memory 110, and a communications interface 114. Memory 110 stores:

an operating system 112;

an Internet communications manager program 116;

a bytecode program verifier 120 for verifying whether or not a specified program satisfies certain predefined integrity criteria;

a bytecode program interpreter 122 for executing application programs;

a class loader 124, which loads object classes into a user's address space and utilizes the bytecode program verifier to verify the integrity of the methods associated with each loaded object class;

at least one class repository 126, for locally storing object classes 128 in use and/or available for use by user's of the computer 102;

at least one object repository 130 for storing objects 132, which are instances of objects of the object classes stored in the object repository 126.

In the preferred embodiment the operating system 112 is an object oriented multitasking operating system that supports multiple threads of execution within each defined address space.

The bytecode program verifier 120 includes a snapshot array 140, a status array 142, and other data structures that will be described in more detail below.

The class loader 124 is typically invoked when a user first initiates execution of a procedure, requiring that an object of the appropriate object class be generated. The class loader 124 loads in the appropriate object class and calls the bytecode program verifier 120 to verify the integrity of all the bytecode programs in the loaded object class. If all the methods are successfully verified an object instance of the object class is generated, and the bytecode interpreter 122 is invoked to execute the user requested procedure, which is typically called a method. If the procedure requested by the user is not a bytecode program and if execution of the non-bytecode program is allowed (which is outside the scope of the present document), the program is executed by a compiled program executer (not shown).

The class loader is also invoked whenever an executing bytecode program encounters a call to an object method for an object class that has not yet been loaded into the user's address space. Once again the class loader 124 loads in the appropriate object class and calls the bytecode program verifier 120 to verify the integrity of all the bytecode programs in the loaded object class. In many situations the object class will be loaded from a remotely located

computer, such as one of the servers 104 shown in FIG. 1. If all the methods in the loaded object class are successfully verified, an object instance of the object class is generated. and the bytecode interpreter 122 is invoked to execute the called object method.

FIG. 2 shows the data structure 200 in a preferred embodiment of the present invention for an object A-01 of class A. An object of object class A has an object handle 202 that includes a pointer 204 to tho methods for the object and a pointer 206 to a data array 208 for the object.

The pointer 204 to the object's methods is actually an indirect pointer to the methods of the associated object class. More particularly, the method pointer 204 points to the Virtual Function Table (VFT) 210 for the object's object class. Each object class has a VFT 210 that includes (A) pointers 212 to each of the methods 214 of the object class, (B) one or more pointers 215 to methods 216 associated with superclasses of class A, and (C) a pointer 217 to a special Class Object 218.

Referring to FIGS. 1 and 2, in the preferred embodiment, the methods in an object class to be loaded are bytecode programs, which when interpreted will result in a series of executable instructions. A listing of all the source code bytecode instructions in the JAVA instruction set is provided in Table 1. The JAVA bytecode instruction set is characterized by bytecode instructions that are data type specific. Specifically, the JAVA instruction set distinguishes the same basic operation on different primitive data types by designating separate opcodes. Accordingly, a plurality of bytecodes are included within instruction set to perform the same basic function (for example to add two numbers), with each such bytecode being used to process only data of a corresponding distinct data type. In addition, the JAVA instruction set is notable for instructions not included. For instance, there are no instructions in the JAVA bytecode language for converting numbers into object references. These restrictions on the JAVA bytecode instruction set help to ensure that any bytecode program which utilizes data in a manner consistent with the data type specific instructions in the JAVA instruction set will not violate the integrity of a user's computer system.

In the preferred embodiment, the available data types are integer, long integer, single precision floating point, double precision floating point, handles (sometimes herein called objects or object references), and return addresses (pointers to virtual machine code). Additional data types are arrays of integers, arrays of long integers, arrays of single precision floating point numbers, arrays of double precision floating point numbers, arrays of handles, arrays of booleans, arrays of bytes (8-bit integers), arrays of short integers (16 bit signed integer), and arrays of unicode characters.

The "handle" data type includes a virtually unlimited number of data subtypes because each handle data type includes an object class specification as part of the data type. In addition, constants used in programs are also data typed, with the available constant data types in the preferred embodiment comprising the data types mentioned above, plus class, fieldref, methodref, string, and Asciz, all of which represent two or more bytes having a specific purpose.

The few bytecodes that are data type independent perform stack manipulation functions such as (A) duplicating one or more words on the stack and placing them at specific locations within the stack, thereby producing more stack items of known data type, or (B) clearing one or more items from the stack. A few other data type independent bytecodes do not utilize any words on the stack and leave the stack unchanged, or add words to the stack without utilizing any of the words previously on the stack. These bytecodes do not have any data type restrictions with regard to the stack contents prior to their execution, and all but a few modify the stack's contents and thus affect the program verification process.

The second computer node 104, assumed here to be configured as a file or other information server, includes a central processing unit 150, a user interface 156, memory 154, and a other communication interface 158 that connects the second computer node to the computer communication network 120. Memory 154 stores programs 163, 164, 166 for execution by the processor 150 and/or distribution to other computer nodes.

The first and second computer nodes 102 and 104 may utilize different computer platforms and operating systems 112, 160 such that object code programs executed on either one of the two computer nodes cannot be executed on the other. For instance, the server node 104 might be a Sun Microsystems computer using a Unix operating system while the user workstation node 102 may be an IBM compatible computer using an 80486 microprocessor and a Microsoft DOS operating system. Furthermore, other user workstations coupled to the same network and utilizing the same server 104 might use a variety of different computer platforms and a variety of operating systems.

In the past, a server 104 used for distributing software on a network having computers of many types would store distinct libraries of software for each of the distinct computer platform types (e.g., Unix, Windows, DOS, Macintosh, etc.). Thus, different versions of the same computer program might be stored in each of the libraries. However, using the present invention, many computer programs could be distributed by such a server using just a single, bytecode version of the program.

The bytecode verifier 120 is an executable program which verifies operand data type compatibility and proper stack manipulations in a specified bytecode (source) program 214 prior to the execution of the bytecode program by the processor 106 under the control of the bytecode interpreter 122. Each bytecode program has an associated verification status value that is True if the program's integrity is verified by the bytecode verifier 120, and it otherwise set to False.

During normal execution of programs using languages other than the Java bytecode language, the interpreter must continually monitor the operand stack for overflows (i.e., adding more data to the stack than the stack can store) and underflows (i.e., attempting to pop data off the stack when the stack is empty). Such stack monitoring must normally be performed for all instructions that change the stack's status (which includes most all instructions). For many programs, stack monitoring instructions executed by the interpreter account for approximately 80% of the execution time of an interpreted computed program.

For many purposes, particularly the integrity of downloaded computer programs, the Internet is a "hostile environment." A downloaded bytecode program may contain errors involving the data types of operands not matching the data type restrictions of the instructions using those operands, which may cause the program to be fail during execution. Even worse, a bytecode program might attempt to create object references (e.g., by loading a computed number into the operand stack and then attempting to use the computed number as an object handle) and to thereby breach the security and/or integrity of the user's computer.

Use of the bytecode verifier 120 in accordance with the present invention enables verification of a bytecode program's integrity and allows the use of an interpreter 122 which does not execute the usual stack monitoring instructions during program execution, thereby greatly accelerating the program interpretation process.

The Bytecode Program Verifier

Referring now to FIG. 3, the bytecode program verifier 120 (often called the "verifier") uses a few temporary data structures to store information it needs while verifying a specified bytecode program 300. In particular, the verifier 120 uses a set of data structures 142 for representing current stack and register status information, and a snapshot data structure 140 for representing the status of the virtual stack and registers just prior to the execution of each instruction in the program being verified. The current status data structures 142 include: a stack size indicator, herein called the stack counter 301, a virtual stack 302 that indicates the data types of all items in the virtual operand stack, a virtual register array 304 that indicates the data types of all items in the virtual registers, and a "jsr" bit vector array 306 that stores zero or more bit vectors associated with the zero or more subroutine calls required to reach the instruction currently being processed.

The stack counter 301, which indicates the number of stack elements that are currently in use (i.e., at the point in the method associated with the instruction currently being analyzed), is updated by the verifier 120 as it keeps track of the virtual stack manipulations so as to reflect the current number of virtual stack entries.

The virtual stack 302 stores data type information regarding each datum that will be stored by the bytecode program 300 in the virtual operand stack during actual execution of the program. In the preferred embodiment, the virtual stack 302 is used in the same way as a regular stack, except that instead of storing actual data and constants, the virtual stack 302 stores a data type indicator value for each datum that will be stored in the operand stack during actual execution of the program. Thus, for instance, if during actual execution the stack were to store three values:

HandleToObjectA
5
1

the corresponding virtual stack entries will be

R;Class A;initialized
I
I

where "R" in the virtual stack indicates an object reference, "Class A" indicates that class or type of the referenced object is "A", "initialized" indicates that the referenced object is an initialized object, and each "I" in the virtual stack indicates an integer. Furthermore, the stack counter 301 in this example would store a value of 3, corresponding to three values being stored in the virtual stack 302.

Data of each possible data type is assigned a corresponding virtual stack marker value, for instance: integer (I), long integer (L), single precision floating point number (F), double precision floating point number (D), byte (B), short (S), and object reference (R). The marker value for an object reference includes a value (e.g., "Class A") indicating the object type and a flag indicating if the object has been initialized. If this is an object that has been created by the current method, but has not yet been initialized, the marker value for the object reference also indicates the program location of the instruction that created the object instance being referenced.

The virtual register array 304 serves the same basic function as the virtual stack 302. That is, it is used to store data type information for registers used by the specified bytecode program. Since data is often transferred by programs between registers and the operand stack, the bytecode instructions performing such data transfers and otherwise using registers can be checked to ensure that the data values in the registers accessed by each bytecode instruction are consistent with the data type usage restrictions on those bytecode instructions.

The structure and use of the jsr bit vector array 306 will be described below in the discussion of the handling of subroutine jumps and returns.

While processing the specified bytecode program, for each datum that would be popped off the stack for processing by a bytecode instruction, the verifier pops off the same number of data type values off the virtual stack 302 and compares the data type values with the data type requirements of the bytecode. For each datum that would be pushed onto the stack by a bytecode instruction, the verifier pushes onto the virtual stack a corresponding data type value.

One aspect of program verification in accordance with present invention is verification that the number of the operands in the virtual stack 302 is identical every time a particular instruction is executed, and that the data types of operands in the virtual stack are compatible. If a particular bytecode instruction can be immediately preceded in execution by two or more different instructions, then the status of the virtual stack immediately after processing of each of those different predecessor instructions must be compared. Usually, at least one of the different preceding instructions will be a conditional or unconditional jump or branch instruction. A corollary of the above "stack consistency" requirement is that each program loop must not result in a net addition or reduction in the number of operands stored in the operand stack.

The stack snapshot array 140 is used to store "snapshots" of the stack counter 301, virtual stack 302, virtual register array 304 and jsr bit vector array 306. A separate snapshot 310 is stored for every instruction in the bytecode program. Each stored stack snapshot includes a "changed" flag 320, a stack counter 321, a stack status array 322, a register status array 324 and a variable length jsr bit vector array 326. The jsr bit vector array 326 is empty except for instructions that can only be reached via one or more jsr instructions.

The changed flag 320 is used to determine which instructions require further processing by the verifier, as will be explained below. The stack counter 321, stack status array 322, register status array 324, and jsr bit vector array 326 are based on the values stored in the data structures 301,302, 304 and 306 at corresponding points in the verification process.

The snapshot storage structure 140 furthermore stores instruction addresses 328 (e.g., the absolute or relative address of each target instruction). Instruction addresses 328 are used by the verifier to make sure that no jump or branch instruction has a target that falls in the middle of a bytecode instruction.

As was described previously, the bytecode program 300 includes a plurality of data type specific instructions, each of which is evaluated by the verifier 120 of the present invention.

Referring now to FIGS. 4A–4F, and Table 2, the execution of the bytecode verifier program 120 will be described in detail. Table 2 lists a pseudocode representation of the verifier program. The pseudocode used in Table 2 utilizes universal computer language conventions. While the pseudocode employed here has been invented solely for the purposes of this description, it is designed to be easily understandable by any computer programmer skilled in the art.

As shown in FIG. 4A, a selected class file containing one or more bytecode methods is loaded (350) into the bytecode verifier 120 for processing. The verifier first performs a number of "non-bytecode" based tests (352) on the loaded class, including verifying:

the class file's format;

that the class is not a subclass of a "final" class;

that no method in the class overrides a "final" method in a superclass;

that each class, other than "Object," has a superclass; and

that each class reference, field reference and method reference in the constant pool has a legal name, class and type signature

If any of these initial verification tests fail, an appropriate error message is displayed or printed, and the verification procedure exits with an abort return code (354).

Next, the verification procedure checks to see if all bytecode methods have been verified (356). If so, the procedure exits with a success return code (358). Otherwise, it selects a next bytecode method in the loaded object class file that requires verification (360).

The code for each method includes the following information:

the maximum stack space needed by the method;

the maximum number of registers used by the method;

the actual bytecodes for executing the method;

a table of exception handlers.

Each entry in the exception handlers tables gives a start and end offset into the bytecodes, an exception type, and the offset of a handler for the exception. The entry indicates that if an exception of the indicated type occurs within the code indicated by the starting and ending offsets, a handler for the exception will be found at the given handler offset.

After selecting a method to verify, the verifier initializes a number of data structures (362), including the stack counter 301, virtual stack 302, virtual register array 304, jsr bit vector array 306, and the snapshot array 140. The snapshot array is initialized as follows. The snapshot for the first instruction of the method is initialized to indicate that the stack is empty and the registers are empty except for data types indicated by the method's type signature, which indicates the initial contents of the registers. The snapshots for all other instructions are initialized to indicate that the instruction has not yet been visited.

In addition, the "changed" bit for the first instruction of the program is set, and a flag called VerificationSuccess is set to True (364). If the VerificationSuccess flag is still set to True when the verification procedure is finished (368), that indicates that the integrity of the method has been verified. If the VerificationSuccess flag is set to False when the verification procedure is finished, the method's integrity has not been verified, and therefore an error message is displayed or printed, and the verification procedure exits with an abort return code (354).

After these initial steps, a data flow analysis is performed on the selected method (366). The details of the data flow analysis, which forms the main part of the verification procedure, is discussed below with reference to FIG. 4B.

In summary, the verification procedure processes each method of the loaded class file until either all the bytecode methods are successfully verified, or the verification of any one of the methods fails.

Data Flow Analysis of Method

Referring to FIG. 4B and the corresponding portion of Table 2, the data flow analysis of the selected method is completed (382) when there are no instructions whose changed bit is set (380). Detection of any stack or register usage error during the analysis causes the VerificationSuccess flag to set to False and for the analysis to be stopped (382).

If there is at least one instruction whose changed bit is set (380), the procedure selects a next instruction whose changed bit is set (384). Any instruction whose changed bit is set can be selected.

The analysis of the selected instruction begins with the pre-existing snapshot for the selected instruction being loaded into the stack counter, virtual stack and the virtual register array, and jsr bit vector array, respectively (386). In addition, the changed bit for the selected instruction is turned off (386).

Next, the effect of the selected instruction on the stack and registers is emulated (388). More particularly, four types of "actions" performed by bytecode instructions are emulated and checked for integrity: stack pops, stack pushes, reading data from registers and writing data to registers. The detailed steps of this emulation process are described next with reference to FIGS. 4C–4G.

Referring to FIG. 4C, if the selected instruction pops data from the stack (450), the stack counter 301 is inspected (452) to determine whether there is sufficient data in the stack to satisfy the data pop requirements of the instruction. If the operand stack has insufficient data (452) for the current instruction, that is called a stack underflow, in which case an error signal or message is generated (454) identifying the place in the program that the stack underflow was detected. In addition, the verifier will then set a VerificationSuccess flag to False and abort (456) the verification process. If no stack underflow condition is detected, the verifier will compare (458) the data type code information previously stored in the virtual stack with the data type requirements (if any) of the currently selected instruction. For example, if the opcode of the instruction being analyzed calls for an integer add of a value popped from the stack, the verifier will compare the operand information of the item in the virtual stack which is being popped to make sure that is of the proper data type, namely integer. If the comparison results in a match, then the verifier deletes (460) the information from the virtual stack associated with the entry being popped and updates the stack counter 301 to reflect the number of entries popped from the virtual stack 302.

If a mismatch is detected (458) between the stored operand information in the popped entry of the virtual stack 302 and the data type requirements of the currently selected instruction, then a message is generated (462) identifying the place in the bytecode program where the mismatch occurred. The verifier will then set the VerificationSuccess flag to False and abort (456) the verification process. This completes the stack pop verification process.

Referring to FIG. 4D, if the currently selected instruction pushes data onto the stack (470), the stack counter is inspected (472) to determine whether there is sufficient room in the stack to store the data the selected instruction will push onto the stack. If the operand stack has insufficient room to store the data to be pushed onto the stack by the current instruction (472), that is called a stack overflow, in which case an error signal or message is generated (474) identifying the place in the program that the stack underflow was detected. In addition, the verifier will then set the VerificationSuccess flag to False and abort (476) the verification process.

If no stack overflow condition is detected, the verifier will add (478) an entry to the virtual stack indicating the type of data (operand) which is to be pushed onto the operand stack (during the actual execution of the program) for each datum to be pushed onto the stack by the currently selected instruction. This information is derived from the data type specific opcodes utilized in the bytecode program of the preferred embodiment of the present invention, the prior contents of the stack and the prior contents of the registers. The verifier also updates the stack counter 301 to reflect the added entry or entries in the virtual stack 302. This completes the stack push verification process.

Referring to FIG. 4E, if the currently selected instruction reads data from a register (510), the verifier will compare (512) the data type code information previously stored in the corresponding virtual register with the data type requirements (if any) of the currently selected instruction. For object handles, data type checking takes into account object class inheritance (i.e., a method that operates on an object of a specified class will can also operate on an object of any subclass of the specified class). If a mismatch is detected (512) between the data type information stored in the virtual register and the data type requirements of the currently selected instruction, then a message is generated (514) identifying the place in the bytecode program where the mismatch occurred. The verifier will then set the VerificationSuccess flag to False and abort (516) the verification process.

The verifier also checks to see if the register accessed by the currently selected instruction has a register number higher than the maximum register number for the method being verified (518). If so, a message is generated (514) identifying the place in the bytecode program where the register access error occurred. The verifier will then set the VerificationSuccess flag to False and abort (516) the verification process.

If the currently selected instruction does not read data from a register (510) or the data type comparison at step 512 results in a match and the registered accessed is within the range of register numbers used by the method being verified (518), then the verifier continues processing the currently selected instruction at step 520.

Referring to FIG. 4F, if the currently selected instruction stores data into a register (520), then the data type associated with the selected bytecode instruction is stored in the virtual register (522).

The verifier also checks to see if the register(s) to be written by the currently selected instruction has(have) a register number higher than the maximum register number for the method being verified (523). If so, an error message is generated (526) identifying the place in the bytecode program where the register access error occurred. The verifier will then set the VerificationSuccess flag to False and abort (528) the verification process.

In addition, the instruction emulation procedure updates the jsr bit vector array 306 as follows. The jsr bit vector array 306 includes a separate bit vector for each subroutine level. Thus, if the current instruction is in a subroutine nested four levels deep, there will be four active jsr bit vectors in the array 306. If the current instruction is in a subroutine that is the target of a jsr instruction (i.e., a jump to subroutine instruction), for each subroutine level applicable to the current instruction, the corresponding jsr bit vector is updated to indicate the register(s) accessed or modified by the current instruction (540, FIG. 4G). The set of "marked" registers in each jsr bit vector can only be increased, not decreased, by step 540.

At this point the procedure for emulating one instruction is complete.

Referring back now to FIG. 4B, if the instruction emulation resulted in the detection of an error, the verification process is halted (382). Otherwise, the next step (390) is to determine the selected instruction's set of successor instructions. The "successor instructions" are defined to be all instructions that might be executed next after the currently selected instruction. The set of all successor instructions includes:

(A) the next instruction in the method, if the current instruction is not an unconditional goto, a return, or a throw;

(B) the target of a conditional or unconditional branch;

(C) all exception handlers for this instruction; and

(D) when the current instruction is a subroutine return instruction, the instructions immediately following all jsr's that target the called subroutine.

It is noted that the last instruction of most exception handlers is a "goto"instruction. More generally, the successor instruction for the end of an exception handler is simply the successor instruction for the last instruction of the exception handler.

As part of the successor instruction determination process, the verifier also checks to see if the program can simply "fall off" the current instruction (i.e., without having a legal next instruction. If so, this is a fatal error and the VerificationSuccess flag is set to False and the verification procedure is terminated (382).

SnapShot Merger

After the successor instruction determination step (390), the verifier next merges the current stack counter 301, virtual stack 302, virtual register array 304 and jsr bit vector arrays 304 into the SnapShots of each of the successor instructions (392). This merger is performed separately for each successor instruction. There are a number of "special" cases requiring special handling of the status and snapshot merger process.

For instance, if a successor instruction is an exception handler, the Stack Status portion of the SnapShot of the successor instruction is defined to contain a single object of the exception type indicated by the exception handler information (i.e., the stored data type for the first virtual stack element indicates the object type of the exception handler), and furthermore the stack counter of the SnapShot of the successor instruction is set to a value of 1.

If the SnapShot for a successor instruction indicates that it has never before been "visited" (i.e., it is empty), the stack counter 301, virtual stack 302, virtual register array 304 and jsr bit vector array 306 are copied into the SnapShot for the successor instruction.

Otherwise, when the instruction has been visited before, the snapshot merger is handled as follows. If the stack counter in the Status Array does not match the stack counter in the existing SnapShot, or the two stacks are not identical with regard to data types, except for differently typed object handles, the VerificationSuccess flag is set to False and the verification process is aborted. Otherwise, the virtual stack

302 and virtual register array 304 values are merged into the values of the successor instruction's existing SnapShot as follows.

If two corresponding virtual stack elements or two corresponding virtual register elements contain different object handles, the specified data type for the stack or register element in the snapshot is replaced with the closest common ancestor (i.e., superclass) of the two handle types. If two corresponding virtual register elements contain different data types (other than handles), the data type for the register element in the updated SnapShot is denoted as "unknown" (i.e., unusable). If two corresponding stack elements contain different data types (other than handles), that is flagged by the verifier as an error.

However, if the successor instruction is the instruction immediately after a "jsr" instruction and the current instruction is a "ret" instruction the above rules for merging virtual register status information are replaced with the following rule:

1) for any register that the corresponding jsr bit vector (i.e., the jsr bit vector for the current instruction that corresponds to the successor jsr instruction) indicates that the subroutine has accessed or modified, update the successor instruction's virtual register SnapShot data to use the data type of the virtual register at the time of the return (i.e., use the virtual register data type information in the corresponding element of the virtual register array 304);

2) for all other registers, update the successor instruction's virtual register SnapShot data to use the data type of the register at the time of the preceding jsr instruction (i.e., copy the virtual register data type information from the corresponding element of the virtual register array in the preceding jsr's instruction's SnapShot).

The snapshot merger procedure also copies the current jsr bit vectors 306 into the SnapShot of the successor instructions only to the extent that those successor instructions are inside the same subroutines as the current instruction.

Finally, after the merger of the current verification status information into the SnapShot of each successor instruction has been performed, the changed bit for the successor instruction is set only if the merging of the virtual stack and register verification status values caused any change to the successor instruction's SnapShot (394).

Thus, the analysis of each selected instruction can cause the changed bit of one or more other instructions to be set. The data flow analysis continues until there are no instructions whose changed bit is set (380). Due to the fact that some instructions are the successor instructions for multiple other instructions, many instructions may be analyzed two or more times by the data flow analysis procedure before the data flow analysis of the method is completed.

Verification Considerations For Exception Handlers

An exception handler is a routine that protects a specified set of program code, called a protected code block. The exception handler is executed whenever the applicable exception gets thrown during execution of the corresponding protected code.

As indicated above, the Stack Status portion of the SnapShot for the first instruction of the exception handler contains a single object of the exception type indicated by the exception handler information (i.e., the stored data type for the first virtual stack element indicates the object type of the exception handler), and further more the stack counter of the SnapShot of the instruction is set to a value of 1.

The virtual register information of the SnapShot for the exception handler's first instruction contains data type values only for registers whose use is consistent throughout the protected code, and contains "unknown" indicators for all other registers used by the protected code.

Verification Considerations for "Finally" Code Blocks

The following program:

```
try     {
            startFaucet();
            waterLawn();
}finally {
            stopFaucet()
}
```

ensures that the faucet is turned off, even if an exception occurs while starting the faucet or watering the lawn. The code inside the bracket after the word "try" is called the protected code. The code inside the brackets after the word "finally" is called the cleanup code. The cleanup code is guaranteed to be executed, even if the protected code does a "return" out of the function, or contains a break or continue to code outside the try/finally code, or experiences an exception.

In the Java bytecode language, the "finally" construct is implemented using the exception handling facilities, together with a "jsr" (jump to subroutine) instruction and "ret" (return from subroutine) instruction. The cleanup code is implemented as a subroutine. When it is called, the top item on the stack will be the return address; this return address is saved in a register. A "ret" is placed at the end of the cleanup code to return to whatever code called the cleanup.

To implement the "finally" feature, a special exception handler is set up for the protected code which catches all exceptions. This exception handler: (1) saves any exception that occurs in a register, (2) executes a "jsr" to the cleanup code, and (3) upon return from the cleanup code, re-throws the exception.

If the protected code has a "return" instruction that when executed will cause a jump to code outside the protected code, the interpreter performs the following steps to execute that instruction: (1) it saves the return value (if any) in a register, (2) executes a "jsr" to the cleanup code, and (3) upon return from the cleanup code, returns the value saved in the register.

Breaks or continue instructions inside the protected code that go outside the protected code are compiled into bytecodes that include a "jsr" to the cleanup code before performing the associated "goto" function. In addition, there must be a "jsr" instruction at the end of the protected code.

The jsr bit vector array and corresponding SnapShot data, as discussed above, enable the successful verification of bytecode programs that contain "finally" constructs. Due to the provision of multiple jsr bit vectors, even multiply-nested cleanup code can be verified.

Verification Considerations for New Object Formation and Initialization

Creating a usable object in the bytecode interpreter is a multi-step process. A typical bytecode sequence for creating and initializing an object, and leaving it on top of the stack is:

```
new <myClass>                /*   allocate uninitialized space     */
dup                          /*   duplicate object on the stack */
<instructions for pushing arguments onto the stack>
invoke myClass.<init>        /*   initialize     */
```

The myClass initialization method, myClass.<init>, sees the newly initialized object as its argument in register 0. It must either call an alternative myClass initialization method or call the initialization method of a superclass of the object before it is allowed to do anything else with the object.

To prevent the use of uninitialized objects, and to prevent objects from being initialized more than once, the bytecode verifier pushes a special data type on the stack as the result of the opcode "new":

R;ObjClass;uninitialized;creationstep

The instruction number (denoted above as "creationstep") needs to be stored as part of the special data type since there may be multiple instances of a not-yet initialized data type in existence at one time. This special data type indicates the instruction in which the object was created and the class type of the uninitialized object created. When an initialization method is called on that object, all occurrences of the special type on the virtual stack and in the virtual registers (i.e., all virtual stack and virtual registers that have the identical data type, including the identical object creation instruction) are replaced by the appropriate, initialized data type:

R;ObjClass;initialized

During verification, the special data type for uninitialized objects is an illegal data type for any bytecode instruction to use, except for a call to an object initialization method for the appropriate object class. Thus, the verifier ensures that an uninitialized object cannot be used until it is initialized.

Similarly, the initialized object data type is an illegal data type for a call to an object initialization method. In this way the verifier ensures that an object is not initialized more than once.

One special check that the verifier must perform during the data flow analysis is that for every backwards branch, the verifier checks that there are no uninitialized objects on the stack or in a register. See steps 530, 532, 534, 536 in FIG. 4F. In addition, there may not be any uninitialized objects in a register in code protected by an exception handler or a finally code block. See steps 524, 526, 528 in FIG. 4F. Otherwise, a devious piece of code could fool the verifier into thinking it had initialized an object when it had, in fact, initialized an object created in a previous pass through the loop. For example, an exception handler could be used to indirectly perform a backwards branch.

Class Loader and Bytecode Interpreter

Referring to flow chart in FIG. 5 and Table 3, the execution of the class loader 124 and bytecode interpreter 122 will be described. Table 3 lists a pseudocode representation of the class loader and bytecode interpreter.

The class loader 124 is typically invoked when a user first initiates execution of a procedure, requiring that an object of the appropriate object class be generated. The class loader 124 loads in the appropriate object class file (560) and calls the bytecode program verifier 120 to verify the integrity of all the bytecode programs in the loaded object class (562). If the verifier returns a "verification failure" value (564), the attempt to execute the specified bytecode program is aborted by the class loader (566).

If all the methods are successfully verified (564) an object instance of the object class is generated, and the bytecode interpreter 122 is invoked (570) to execute the user requested procedure, which is typically called a method. The bytecode interpreter of the present invention does not perform (and does not need to perform) any operand stack overflow and underflow checking during program execution and also does not perform any data type checking for data stored in the operand stack during program execution. These conventional stack overflow, underflow and data type checking operations can be skipped by the present invention because the verifier has already verified that errors of these types will not be encountered during program execution.

The program interpreter of the present invention is especially efficient for execution of bytecode programs having instruction loops that are executed many times, because the operand stack checking instructions are executed only once for each bytecode in each such instruction loop in the present invention. In contrast, during execution of a program by a conventional interpreter, the interpreter must continually monitor the operand stack for overflows (i.e., adding more data to the stack than the stack can store) and underflows (i.e., attempting to pop data off the stack when the stack is empty). Such stack monitoring must normally be performed for all instructions that change the stack's status (which includes most all instructions). For many programs, stack monitoring instructions executed by the interpreter account for approximately 80% of the execution time of an interpreted computed program. As a result, the interpreter of the present invention will often execute programs at two to ten times the speed of a conventional program interpreter running on the same computer.

Alternate Embodiments

The foregoing descriptions of specific embodiments of the present invention have been presented for purposes of illustration and description. They are not intended to be exhaustive or to limit the invention to the precise forms disclosed, and obviously many modifications and variations are possible in light of the above teaching. The embodiments were chosen and described in order to best explain the principles of the invention and its practical application, to thereby enable others skilled in the art to best utilize the invention and various embodiments with various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the claims appended hereto and their equivalents.

TABLE 1

BYTECODES IN JAVA LANGUAGE

INSTRUCTION NAME	SHORT DESCRIPTION
nop	no operation
aconst_null	push null object
iconst_m1	push integer constant-1
iconst_0	push integer constant 0
iconst_1	push integer constant 1
iconst_2	push integer constant 2
iconst_3	push integer constant 3
iconst_4	push integer constant 4
iconst_5	push integer constant 5
lconst_0	push long 0L
lconst_1	push long 1L
fconst_0	push float constant 0.0
fconst_1	push float constant 1.0
fconst_2	push float constant 2.0
dconst_0	push double float constant 0.0d
dconst_1	push double float constant 1.0d
bipush	push byte-sized value

TABLE 1-continued

BYTECODES IN JAVA LANGUAGE

INSTRUCTION NAME	SHORT DESCRIPTION
sipush	push two-byte value
ldc	load a constant from constant table (1 byte index)
ldc_w	load a constant from constant table (2 byte index)
ldc2_w	load a 2-word constant...
iload	load local integer variable
lload	load local long variable
fload	load local floating variable
dload	load local double variable
aload	load local object variable
iload_0	load local integer variable #0
iload_1	load local integer variable #1
iload_2	load local integer variable #2
iload_3	load local integer variable #3
lload_0	load local long variable #0
lload_1	load local long variable #1
lload_2	load local long variable #2
lload_3	load local long variable #3
fload_0	load local float variable #0
fload_1	load local float variable #1
fload_2	load local float variable #2
fload_3	load local float variable #3
dload_0	load lcl double float variable #0
dload_1	load lcl double float variable #1
dload_2	load lcl double float variable #2
dload_3	load lcl double float variable #3
aload_0	load local object variable #0
aload_1	load local object variable #1
aload_2	load local object variable #2
aload_3	load local object variable #3
iaload	load from array of integer
laload	load from array of long
faload	load from array of float
daload	load from array of double
aaload	load from array of object
baload	load from array of (signed) bytes
caload	load from array of chars
saload	load from array of (signed) shorts
istore	store local integer variable
lstore	store local long variable
fstore	store local float variable
dstore	store local double variable
astore	store local object variable
istore_0	store local integer variable #0
istore_1	store local integer variable #1
istore_2	store local integer variable #2
istore_3	store local integer variable #3
lstore_0	store local long variable #0
lstore_1	store local long variable #1
lstore_2	store local long variable #2
lstore_3	store local long variable #3
fstore_0	store local float variable #0
fstore_1	store local float variable #1
fstore_2	store local float variable #2
fstore_3	store local float variable #3
dstore_0	store lcl double float variable #0
dstore_1	store lcl double float variable #1
dstore_2	store lcl double float variable #2
dstore_3	store lcl double float variable #3
astore_0	store local object variable #0
astore_1	store local object variable #1
astore_2	store local object variable #2
astore_3	store local object variable #3
iastore	store into array of int
lastore	store into array of long
fastore	store into array of float
dastore	store into array of double float
aastore	store into array of object
bastore	store into array of (signed) bytes
castore	store into array of chars
sastore	store into array of (signed) shorts
pop	pop top element
pop2	pop top two elements
dup	dup top element

TABLE 1-continued

BYTECODES IN JAVA LANGUAGE

INSTRUCTION NAME	SHORT DESCRIPTION
dup_x1	dup top element. Skip one
dup_x2	dup top element. Skip two
dup2	dup top two elements.
dup2_x1	dup top 2 elements. Skip one
dup2_x2	dup top 2 elements. Skip two
swap	swap top two elements of stack.
iadd	integer add
ladd	long add
fadd	floating add
dadd	double float add
isub	integer subtract
lsub	long subtract
fsub	floating subtract
dsub	floating double subtract
imul	integer multiply
lmul	long multiply
fmul	floating multiply
dmul	double float multiply
idiv	integer divide
ldiv	long divide
fdiv	floating divide
ddiv	double float divide
irem	integermod
lrem	long mod
frem	floating mod
drem	double float mod
ineg	integer negate
lneg	long negate
fneg	floating negate
dneg	double float negate
ishl	shift left
lshl	long shift left
ishr	shift right
lshr	long shift right
iushr	unsigned shift right
lushr	long unsigned shift right
iand	boolean and
land	long boolean and
ior	boolean or
lor	long boolean or
ixor	boolean xor
lxor	long boolean xor
iinc	increment lcl variable by constant
i2l	integer to long
i2f	integer to float
i2d	integer to double
l2i	long to integer
l2f	long to float
l2d	long to double
f2i	float to integer
f2l	float to long
f2d	float to double
d2i	double to integer
d2l	double to long
d2f	double to float
int2byte	integer to byte
int2char	integer to character
int2short	integer to signed short
lcmp	long compare
fcmpl	float compare. -1 on incomparable
fcmpg	float compare. 1 on incomparable
dcmpl	dbl floating cmp. -1 on incomp
dcmpg	dbl floating cmp. 1 on incomp
ifeq	goto if equal
ifne	goto if not equal
iflt	goto if less than
ifge	goto if greater than or equal
ifgt	goto if greater than
ifle	goto if less than or equal
if_icmpeq	compare top two elements of stack
if_icmpne	compare top two elements of stack
if_icmplt	compare top two elements of stack
if_icmpge	compare top two elements of stack
if_icmpgt	compare top two elements of stack
if_icmple	compare top two elements of stack

TABLE 1-continued

BYTECODES IN JAVA LANGUAGE

INSTRUCTION NAME	SHORT DESCRIPTION
if_acmpeq	compare top two objects of stack
if_acmpne	compare top two objects of stack
goto	unconditional goto
jsr	jump subroutine
ret	return from subroutine
tableswitch	goto (case)
lookupswitch	goto (case)
ireturn	return integer from procedure
lreturn	return long from procedure
freturn	return float from procedure
dreturn	return double from procedure
areturn	return object from procedure
return	return (void) from procedure
getstatic	get static field value.
putstatic	assign static field value
getfield	get field value from object.
putfield	assign field value to object.
invokevirtual	call method, based on object.
invokenonvirtual	call method, not based on object.
invokestatic	call a static method.
invokeinterface	call an interface method
new	Create a new object
newarray	Create a new array of non-objects
anewarray	Create a new array of objects
arraylength	get length of array
athrow	throw an exception
checkcast	error if object not of given type
instanceof	is object of given type?
monitorenter	enter a monitored region of code
monitorexit	exit a monitored region of code
wide	prefix operation.
multianewarray	create multidimensional array
ifnull	goto if null
ifnonnull	goto if not null
goto_w	unconditional goto. 4byte offset
jsr_w	jump subroutine. 4byte offset
breakpoint	call breakpoint handler

TABLE 2

Pseudocode for JAVA Bytecode Verifier

```
Receive Object Class File with one or more bytecode programs to
be verified.
/*  Perform initial checks that do not require inspection of bytecodes  */
If file format of the class file is improper
    {
    Print appropriate error message
    Return with Abort return code
    }
If  (A) any "final" class has a subclass;
    (B) the class is a subclass of a "final" class;
    (C) any method in the class overrides a "final" method in a
        superclass; or
    (D) any class reference, field reference and method reference in the
        constant pool does not have a legal name, class and type
        signature
    {
    Print appropriate error message
    Return with Abort return code
    }
For each Bytecode Method in the Class
    {
    /*  Data-flow analysis is performed on each method of the class
        being verified */
    If: (A) any branch instruction would branch into the middle of an
        instruction,
        (B) any register references access or modify a register having a
        register number higher than the number of registers used by the
        method,
        (C) the method ends in the middle of an instruction,
        (D) any instruction having a reference into the constant pool
```

TABLE 2-continued

Pseudocode for JAVA Bytecode Verifier

```
        does not match the data type of the referenced constant pool
        item,
        (B) any exception handler does not have properly specified
        starting and ending points,
        {
        Print appropriate error message
        Return with Abort return code
        }
Create status data structures: stack counter, stack status array.
    register status array, jsr bit vector array
Create SnapShot array with one SnapShot for every instruction in the
    bytecode program
Initialize SnapShot for first instruction of program to indicate the
    stack is empty and the registers are empty except for data types
    indicated by the method's type signature (i.e., for arguments
    to be passed to the method)
Initialize Snapshots for all other instructions to indicate that the
    instruction has not yet been visited
Set the "changed" bit for the first instruction of the program
Set VerificationSuccess to True
Do Until there are no instructions whose changed bit is set
    {
    Select a next instruction (e.g., in sequential order in program)
        whose changed bit is set
    Load SnapShot for the selected instruction (showing status of
        stack and registers prior to execution of the selected
        instruction) into the stack counter, virtual stack and the
        virtual register array, and jsr bit vector array, respectively.
    Turn off the selected instruction's changed bit
    /*  Emulate the effect of this instruction on the stack and
        registers*/
    Case(Instruction Type):
        {
        Case=Instruction pops data from Operand Stack
            {
            Pop operand data type information from Virtual Stack
        Update Stack Counter
        If Virtual Stack has Underflowed
            {
            Print error message identifying place in program that
                underflow occurred
            Abort Verification
            Return with abort return code
            }
        Compare data type of each operand popped from virtual
            stack with data type required (if any) by the bytecode
            instruction
        If type mismatch
            {
            Print message identifying place in program that data
                type mismatch occurred
            Set VerificationSuccess to False
            Return with abort return code
            }
        }
    Case=Instruction pushes data onto Operand Stack
        {
        Push data type information onto Virtual Stack
        Update stack counter
        If Virtual Stack has Overflowed
            {
            Print message identifying place in program that
                overflow occurred
            Set VerificationSuccess to False
            Return with abort return code
            }
        }
    Case=Instruction uses data stored in a register
        {
        If type mismatch
            {
            Print message identifying place in program that data
                type mismatch occurred
            Set VerificationSuccess to False
            }
        }
    Case=Instruction modifies a register
```

TABLE 2-continued

Pseudocode for JAVA Bytecode Verifier

```
            {
            Update Virtual Register Array to indicate changed register's
                new data type
            If instruction places an uninitialized object in a register and
                the instruction is protected by any exception handler
                (including the special exception handler for a "finally"
                code block)
                {
                Print error message
                Set VerificationSuccess to False
                }
            }
        Case=Backwards Branch
            {
            If Virtual Stack or Virtual Register Array contain any
                uninitialized object data types
                {
                Print error message
                Set VerificationSuccess to False
                }
            }
    } /* EndCase */
    /*  Update jsr bit vector array   */
    If the current instruction is in a subroutine that is the target of a jsr
        {
        For each level of jsr applicable to the current instruction
            {
        Update corresponding jsr bit vector to indicate register(s)
            accessed or modified by the current instruction
        /*  Set of "marked" registers can only be increased, not
            decreased */
            }
        }
    /*  Update all affected SnapShots and changed bits      */
    Determine set of all successor instructions, including:
        (A) the next instruction if the current instruction is not an
            unconditional goto, a return, or a throw,
        (B) the target of a conditional or unconditional branch,
        (C) all exception handlers for this instruction,
        (D) when the current instruction is a return instruction, the
            successor instructions are the instructions immediately
            following all jsr's that target the called subroutine
    If the program can "fall off" the last instruction
        {
        Set VerificationSuccess to False
        Return with Abort return code value
        }
/*  Merge the stack counter, virtual stack, virtual register array and jsr bit
        vector arrays into the SnapShots of each of the successor
        instructions */
    Do for each successor instruction:
        {
        If the successor instruction is the first instruction of an exception
            handler,
            {
            Change the Stack Status portion of the SnapShot of the
                successor instruction to contain a single object of the
                exception type indicated by the exception handler
                information.
            Set stack counter of the SnapShot of the successor
                instruction to 1.
            Performs steps noted below for successor instruction
                handling only with respect to the virtual register array
                and jsr bit vector array.
            }
        If this is the first time the SnapShot for a successor instruction
            has been visited
            {
            Copy the stack counter, virtual stack, virtual register array
                and jsr bit vector array into the SnapShot for the
                successor instruction
            Set the changed bit for the successor instruction
            }
    Else        /*  the instruction has been visited before   */
            {
            If the stack counter in the Status Array does not match the
                stack counter in the existing SnapShot, or the two
```

TABLE 2-continued

Pseudocode for JAVA Bytecode Verifier

```
            stacks are not identical with regard to data types
            (except for differently typed object handles)
            {
            Set VerificationSuccess to False
            Return with Abort return code value
            }
        Merge the Virtual Stack and Virtual Register Array values
            into the values of the existing SnapShot:
            (A) if two corresponding stack elements or two
            corresponding register elements contain different
            object handles, replace the specified data type for the
            stack or register element with the closest common
            ancestor of the two handle types;
            (B) if two corresponding register elements contain
            different data types (other than handles), denote the
            specified data type for the register element in the new
            SnapShot as "unknown" (i.e., unusable);
            (C) follow special merger rules for merging register
            status information when the successor instruction is the
            instruction immediately after a "jsr" instruction and the
            current instruction is a "ret" instruction:
                1)  for any register that the bit vector indicates
                    that the subroutine has accessed or modified,
                    use the data type of the register at the time of
                    the return, and
                2)  for other registers, use the data type of the
                    register at the time of the preceding jsr
                    instruction.
        /*  Note that return, break and continue instructions
            inside a code block protected by a "finally"
            exception handler are treated the same as a "jsr"
            instruction (for a subroutine call to the "finally"
            exception handler) for verification purposes. */
        Copy the jsr bit vectors into the SnapShot of the
            successor instructions only to the extent that those
            successor instructions are inside the same
            subroutines as the current instruction.
        Set the changed bit for each successor instruction for
            which the merging of the stack and register values
            caused any change to the successor instruction's
            SnapShot.
        }
      }     /* End of Do Loop for Successor Instructions */
    }       /* End of Do Loop for Instruction Emulation */
  }         /* End of Loop for Bytecode Methods */
Return (VerificationSuccess)
```

TABLE 3

Pseudocode for Bytecode Class Loader and Interpreter

```
Procedure: ClassLoader (Class, Pgm)
{
If the Class has not already been loaded and verified
    {
    Receive Class
    Call Bytecode Verifier to verify all bytecode programs in the class
    If Not VerificationSuccess
        {
        Print or display appropriate error message
        Return
        }
    }
Interpret and execute Pgm (the specified bytecode program) without
    performing operand stack overflow and underflow checks and without
    performing data type checks on operands stored in operand stack.
}
```

What is claimed is:

1. A method of operating a computer system, the steps of the method comprising:

(A) storing a program in a memory, the program including a sequence of instructions, where each of a multiplicity of said instructions each represents an operation on data of a specific data type; said each instruction having associated data type restrictions on the data type of data to be manipulated by said each instruction;

(B) prior to execution of said program, preprocessing said program by determining whether execution of any instruction in said program would violate said data type restrictions for that instruction and generating a program fault signal when execution of any instruction in said program would violate the data type restrictions for that instruction;

said preprocessing step including:

(B1) storing, for each instruction in said program, a data type snapshot, said data type snapshot including data type information concerning data types associated with data stored in an operand stack and registers by said program immediately prior to execution of the corresponding instruction;

(B2) emulating operation of a selected instruction in the program by: (B2A) analyzing stack and register usage by said selected instruction so as to generate a current data type usage map for said operand stack and registers, (B2B) determining all successor instructions to said selected instruction, (B2C) merging the current data type usage map with the data type snapshot of said determined successor instructions, and (B2D) marking for further analysis each of said determined successor instructions whose data type snapshot is modified by said merging;

(B3) emulating operation of each of said instructions marked for further analysis by performing step B2 on each of those marked instructions and unmarking each said emulated instruction; and

(B4) repeating step B3 until there are no marked instructions;

said step B2A including determining when said stack and register usage by said instruction would violate said data type restrictions for that instruction and generating a program fault signal when execution of said instruction program would violate said data type restrictions.

2. The method of claim 1, said step B2 including determining whether execution of said selected instruction would result in an operand stack underflow or overflow, and whether execution of any loop in said program would result in a net addition or deletion of operands to said operand stack, and for generating a program fault signal when said execution of said selected instruction would result in an operand stack underflow or overflow and when execution of any loop in said program would produce a net addition or deletion of operands to said operand stack.

3. The method of claim 1, including

(C) when said preprocessing of said program results in the generation of no program fault signals, enabling execution of said program;

(D) when said preprocessing of said program results in the generation of a program fault, preventing execution of said program; and

(E) when execution of said bytecode program has been enabled, executing said bytecode program without performing data type checks on operands stored in said operand stack during execution of said bytecode program.

4. The method of claim 1,

said program including at least one object creation instruction and at least one object initialization instruction;

said step 2B including storing in said current usage data map, for each object that would be stored in said operand stack and registers after execution of said selected instruction, a data type value for each uninitialized object that is distinct from a corresponding data type value for the same object after initialization thereof;

said step 2B further including, when said selected instruction is not said at least one object initialization instruction, generating a program fault signal when execution of said selected instruction would access a stack operand or register whose data type corresponds to an uninitialized object.

5. The method of claim 4,

said step 2B further including, when said selected instruction is said at least one object initialization instruction, generating a program fault signal when execution of said selected instruction would access a stack operand or register whose data type corresponds to an initialized object.

6. The method of claim 1,

said program including at least one jump to subroutine (jsr) instruction and at least one subroutine return (ret) instruction located within a subroutine included in said program;

said step B2B including, when the current instruction is said subroutine return instruction, determining each of said successor instructions to be an instruction immediately following a jsr instruction for jumping to said subroutine;

said step B2C including, when the current instruction is said subroutine return instruction, merging the current data type usage map with the data type snapshot of each said determined successor instructions by storing in the data type snapshot for each said successor instruction data type information from said current data type usage map for each register accessed and each register modified by said subroutine and data type information for each other register from the data type snapshot for the jsr instruction immediately preceding said each successor instruction.

7. A computer system, comprising:

memory for storing a program, the program including a sequence of instructions, where each of a multiplicity of said instructions each represents an operation on data of a specific data type; said each instruction having associated data type restrictions on the data type of data to be manipulated by said each instruction;

a data processing unit for executing programs stored in said memory;

a program verifier, stored in said memory, said program verifier including data type testing instructions for determining whether execution of any instruction in said program would violate said data type restrictions for that instruction and generating a program fault signal when execution of any instruction in said program would violate the data type restrictions for that instruction;

said data type testing instructions including:

instructions for storing, for each instruction in said program, a data type snapshot, said data type snapshot including data type information concerning data types associated with data stored in an operand stack and registers by said program immediately prior to execution of the corresponding instruction;

instructions for emulating operation of a selected instruction in the program by: analyzing stack and

register usage by said selected instruction so as to generate a current data type usage map for said operand stack and registers, determining all successor instructions to said selected instruction, merging the current data type usage map with the data type snapshot of said determined successor instructions, and marking for further analysis each of said determined successor instructions whose data type snapshot is modified by said merging;

instructions for emulating operation of each of said instructions marked for further analysis and unmarking each said emulated instruction; and

instructions for continuing to emulate operation of any instructions marked for further analysis until there are no marked instructions;

said data type testing instructions including instructions for determining when said stack and register usage by said instruction would violate said data type restrictions for that instruction and generating a program fault signal when execution of said instruction program would violate said data type restrictions.

8. The computer system of claim 7, including:

program execution enabling instructions that enable execution of said bytecode program only after processing said bytecode program by said bytecode program verifier generates no program fault signals; and

a bytecode program interpreter, coupled to said bytecode program enabling instructions, for executing said bytecode program after processing of said bytecode program by said bytecode program verifier and after said bytecode program enabling instructions enable execution of said bytecode program by said bytecode program interpreter; said bytecode program interpreter including instructions for executing said bytecode program without performing data type checks on operands stored in said operand stack.

9. The computer system of claim 8,

said data type testing instructions, including stack overflow/underflow testing instructions for determining (A) whether execution of said program would result in an operand stack underflow or overflow, and (B) whether execution of any loop in said program would result in a net addition or deletion of operands to said operand stack, and for generating a program fault signal when said execution of said selected instruction would result in an operand stack underflow or overflow and when execution of any loop in said program would produce a net addition or deletion of operands to said operand stack; and

said executing instructions of said bytecode program interpreter including instructions for executing said bytecode program without performing operand stack underflow and overflow checks during execution of said bytecode program.

10. The computer system of claim 7,

said program including at least one object creation instruction and at least one object initialization instruction;

said data type testing instructions including instructions for storing in said current usage data map, for each object that would be stored in said operand stack and registers after execution of said selected instruction, a data type value for each uninitialized object that is distinct from a corresponding data type value for the same object after initialization thereof;

said data type testing instructions further including instructions for generating, when said selected instruction is not said at least one object initialization instruction, a program fault signal when execution of said selected instruction would access a stack operand or register whose data type corresponds to an uninitialized object.

11. The computer system of claim 10,

said data type testing instructions further including instructions for generating, when said selected instruction is said at least one object initialization instruction, a program fault signal when execution of said selected instruction would access a stack operand or register whose data type corresponds to an initialized object.

12. The computer system of claim 7,

said program including at least one jump to subroutine (jsr) instruction and at least one subroutine return (ret) instruction located within a subroutine included in said program;

said data type testing instructions including instructions for determining, when the current instruction is said subroutine return instruction, each of said successor instructions to be an instruction immediately following a jsr instruction for jumping to said subroutine;

said data type testing instructions including instructions for merging, when the current instruction is said subroutine return instruction, the current data type usage map with the data type snapshot of each said determined successor instructions by storing in the data type snapshot for each said successor instruction data type information from said current data type usage map for each register accessed and each register modified by said subroutine and data type information for each other register from the data type snapshot for the jsr instruction immediately preceding said each successor instruction.

13. A computer program product for use in conjunction with a computer system, the computer program product comprising a computer readable storage medium and a computer program mechanism embedded therein, the computer program mechanism comprising:

a program stored in said memory, the program including a sequence of instructions, where each of a multiplicity of said instructions each represents an operation on data of a specific data type; said each instruction having associated data type restrictions on the data type of data to be manipulated by said each instruction;

a program verifier, stored in said memory, said program verifier including data type testing instructions for determining whether execution of any instruction in said program would violate said data type restrictions for that instruction and generating a program fault signal when execution of any instruction in said program would violate the data type restrictions for that instruction;

said data type testing instructions including:

instructions for storing, for each instruction in said program, a data type snapshot, said data type snapshot including data type information concerning data types associated with data stored in an operand stack and registers by said program immediately prior to execution of the corresponding instruction;

instructions for emulating operation of a selected instruction in the program by: analyzing stack and register usage by said selected instruction so as to generate a current data type usage map for said operand stack and registers, determining all successor instructions to said selected instruction, merging the current data type usage map with the data type snapshot of said determined successor instructions, and marking for further analysis each of said determined successor instructions whose data type snapshot is modified by said merging;

instructions for emulating operation of each of said instructions marked for further analysis and unmarking each said emulated instruction; and

instructions for continuing to emulate operation of any instructions marked for further analysis until there are no marked instructions;

said data type testing instructions including instructions for determining when said stack and register usage by said instruction would violate said data type restrictions for that instruction and generating a program fault signal when execution of said instruction program would violate said data type restrictions.

14. The computer program product of claim 13, including:

program execution enabling instructions that enable execution of said bytecode program only after processing said bytecode program by said bytecode program verifier generates no program fault signals; and

a bytecode program interpreter, coupled to said bytecode program enabling instructions, for executing said bytecode program after processing of said bytecode program by said bytecode program verifier and after said bytecode program enabling instructions enable execution of said bytecode program by said bytecode program interpreter; said bytecode program interpreter including instructions for executing said bytecode program without performing data type checks on operands stored in said operand stack.

15. The computer program product of claim 14,

said data type testing instructions including stack overflow/underflow testing instructions for determining (A) whether execution of said program would result in an operand stack underflow or overflow, and (B) whether execution of any loop in said program would result in a net addition or deletion of operands to said operand stack, and for generating a program fault signal when said execution of said selected instruction would result in an operand stack underflow or overflow and when execution of any loop in said program would produce a net addition or deletion of operands to said operand stack; and

said executing instructions of said bytecode program interpreter including instructions for executing said bytecode program without performing operand stack underflow and overflow checks during execution of said bytecode program.

16. The computer program product of claim 13,

said program including at least one object creation instruction and at least one object initialization instruction;

said data type testing instructions including instructions for storing in said current usage data map, for each object that would be stored in said operand stack and registers after execution of said selected instruction, a data type value for each uninitialized object that is distinct from a corresponding data type value for the same object after initialization thereof;

said data type testing instructions further including instructions for generating, when said selected instruction is not said at least one object initialization instruction, a program fault signal when execution of said selected instruction would access a stack operand or register whose data type corresponds to an uninitialized object.

17. The computer program product of claim 16,

said data type testing instructions further including instructions for generating, when said selected instruction is said at least one object initialization instruction, a program fault signal when execution of said selected instruction would access a stack operand or register whose data type corresponds to an initialized object.

18. The computer program product of claim 13,

said program including at least one jump to subroutine (jsr) instruction and at least one subroutine return (ret) instruction located within a subroutine included in said program;

said data type testing instructions including instructions for determining, when the current instruction is said

subroutine return instruction, each of said successor instructions to be an instruction immediately following a jsr instruction for jumping to said subroutine;

said data type testing instructions including instructions for merging, when the current instruction is said subroutine return instruction, the current data type usage map with the data type snapshot of each said determined successor instructions by storing in the data type snapshot for each said successor instruction data type information from said current data type usage map for each register accessed and each register modified by said subroutine and data type information for each other register from the data type snapshot for the jsr instruction immediately preceding said each successor instruction.

* * * * *

EXHIBIT 10

COMBATING VIRUSES HEURISTICALLY

Frans Veldman

ESaSS B.V., Kerkenbos 10-21, 6546 BB Nijmegen, The Netherlands.
Tel: +31 80 787881 · Fax: +31 80 789186

1 INTRODUCTION

Generally speaking, there are two basic methods to detect viruses - specific and generic. Specific virus detection requires the anti-virus program to have some pre-defined information about a specific virus (like a scan string). The anti-virus program must be frequently updated in order to make it detect new viruses as they appear. Generic detection methods however are based on generic characteristics of the virus, so theoretically they are able to detect every virus, including the new and unknown ones.

Why is generic detection gaining importance? There are four reasons:

1) The number of viruses increases rapidly. Studies indicate that the total number of viruses doubles roughly every nine months. The amount of work for the virus researcher increases, and the chances that someone will be hit by one of these unrecognizable new viruses increases also.

2) The number of virus mutants increases. Virus source codes are widely spread and many people can't resist the temptation to experiment with them, creating many slightly modified viruses. These modified viruses may or may not be recognized by the anti-virus product. Sometimes they are but, unfortunately, often they are not.

3) The development of polymorphic viruses. Polymorphic viruses like MTE and TPE are more difficult to detect with virus scanners. It is often months after a polymorphic virus has been discovered before a reliable detection algorithm has been developed. In the meantime many users have an increased chance of being infected by that virus.

4) Viruses directed at a specific organization or company. It is possible for individuals to utilize viruses as weapons. By creating a virus that only works on machines owned by a specific organization or company it is very unlikely that the virus will spread outside the organization. Thus it is very unlikely that any virus scanner will be able to detect the virus before the payload of the virus does its destructive work and reveals itself.

Each of these scenarios demonstrates the fact that virus scanners cannot recognize a virus until the virus has been discovered and analysed by an anti-virus vendor.

These same scenarios do not hold true for generic detectors, therefore many people are becoming more interested in generic anti-virus products. Of the many generic detection methods, heuristic scanning is currently becoming the most important.

2 HEURISTIC SCANNING

One of the most time-consuming tasks that a virus researcher faces is the examination of files. People often send files to researchers because they believe the files are infected by a new virus. Sometimes these files are indeed infected, sometimes not. Every researcher is able to determine very quickly what is going on by loading the suspected file into a debugger. A few seconds is often enough, and many researchers must have asked themselves: "How can I determine this so quickly?"

This question is often difficult to answer. It is just as difficult as describing how one ties his shoelaces. Almost everyone is able to perform this simple task, but few are able to describe it in clear text.

2.1 ARTIFICIAL INTELLIGENCE

One of the many differences between viruses and normal programs is that normal programs typically start searching the command line for options, clearing the screen, etc. Viruses however never search for command line options or clear the screen. Instead they start with a search for other executable files, by writing to the disk, or by decrypting themselves.

A researcher who has loaded the suspected file into a debugger can notice this difference in just a glance. Heuristic scanning is an attempt to put this experience and knowledge into a virus scanner. Heuristic scanning is indeed a kind of artificial intelligence.

The word 'heuristic' means (according to a Dutch dictionary) 'the self-finding' and 'the knowledge to determine something in a methodic way'. An heuristic scanner is a type of automatic debugger or disassembler. The instructions are disassembled and their purposes are determined. If a program starts with the sequence MOV AH,5 INT 13h which is a disk format instruction for the BIOS, this is highly suspicious, especially if the program does not process any command line options or interact with the user.

2.2 SUSPECTED ABILITIES

In reality, heuristics is much more complicated. The heuristic scanners that I am familiar with are able to detect suspicious instruction sequences, like the ability to format a disk, the ability to search for other executables, the ability to remain resident in memory, the ability to issue non-standard or undocumented system calls, etc. Each of these abilities has a value assigned to it. If the total of the values for one program exceeds a pre-defined threshold, the scanner yells "Virus!". A single suspected ability is never enough to trigger the alarm. It is always the combination of the suspected abilities which convince the scanner that the file is a virus.

2.3 HEURISTIC FLAGS

Some scanners set a flag for each suspected ability which has been found in the file being analysed. This makes it easier to explain to the user what has been found.

TbScan for instance recognizes many suspected instruction sequences. Every suspected instruction sequence has a flag assigned to it.

2.4 FLAG DESCRIPTION

F = Suspicious file access. Might be able to infect a file.

R = Relocator. Program code will be relocated in a suspicious way.

A = Suspicious Memory Allocation. The program uses a non-standard way to search for and/or allocate memory.

N = Wrong name extension. Extension conflicts with program structure.

S = Contains a routine to search for executable (.COM or .EXE) files.

= Found an instruction decryption routine. This is common for viruses but also for some protected software.

E = Flexible Entry-point. The code seems to be designed to be linked on any location within an executable file. Common for viruses.

L = The program traps the loading of software. Might be a virus that intercepts program load to infect the software.

D = Disk write access. The program writes to disk without using DOS.

M = Memory resident code. This program is designed to stay in memory.

! = Invalid opcode (non-8088 instructions) or out-of-range branch.

T = Incorrect timestamp. Some viruses use this to mark infected files.

J = Suspicious jump construct. Entry point via chained or indirect jumps. This is unusual for normal software but common for viruses.

? = Inconsistent exe-header. Might be a virus but can also be a bug.

G = Garbage instructions. Contains code that seems to have no purpose other than encryption or avoiding recognition by virus scanners.

U = Undocumented interrupt/DOS call. The program might be just tricky but can also be a virus using a non-standard way to detect itself.

Z = .EXE/.COM determination. The program tries to check whether a file is a .COM or .EXE file. Viruses need to do this to infect a program.

O = Found code that can be used to overwrite/move a program in memory.

B = Back to entry point. Contains code to re-start the program after modifications at the entry-point are made. Very usual for viruses.

K = Unusual stack. The program has a suspicious stack or an odd stack.

TbScan would for instance output the following flags:

VIRUS	HEURISTIC FLAGS
Jerusalem/PLO	FRLMUZ
Backfont	FRALDMUZK
Minsk_Ghost	FELDTGUZB
Murphy	FSLDMTUZO
Ninja	FEDMTUZOBK
Tolbuhin	ASEDMUOB
Yankee_Doodle	FN#ELMUZB

The more flags that are triggered by a file, the more likely it is that the file is infected by a virus. Normal programs rarely trigger even one flag, while at least two flags are required to trigger the alarm. To make it more complicated, not all flags carry the same 'weight'.

3 FALSE POSITIVES

Just like all other generic detection techniques, heuristic scanners sometimes blame innocent programs for being contaminated by a virus. This is called a "False Positive" or "False Alarm".

The reason for this is simple. Some programs happen to have several suspected abilities. For instance, the LOADHI.COM file of QEMM has the following suspected abilities (according to TbScan):

A = Suspicious Memory Allocation. The program uses a non-standard way to search for and/or allocate memory.

M = Memory resident code. This program may be a TSR but also a virus.

U = Undocumented interrupt/DOS call. The program might be just tricky but can also be a virus using a non-standard way to detect itself.

Z = .EXE/.COM determination. The program tries to check whether a file is a .COM or .EXE file. Viruses need to do this to infect a program.

O = Found code that can be used to overwrite/move a program in memory.

All of these abilities are available in LoadHi and the flags are enough to trigger the heuristic alarm of TbScan. As LoadHi is supposed to allocate upper memory, load resident programs in memory, move them to upper memory, etc., all these suspected abilities can easily be explained and verified. However, the scanner is not able to know the intended purpose of the program and, as most of these suspected abilities are often found in viruses, it just describes the LoadHi program as a "possible virus".

3.1 HOW SERIOUS IS THE ISSUE OF FALSE ALARMS?

If an heuristic scanner pops up with a message saying: "This program is able to format a disk and it stays resident in memory" and the program is a resident disk format utility, is this really a false alarm? Actually, the scanner is right. A resident format utility obviously contains code to format a disk and it contains code to stay resident in memory. The heuristic scanner is therefore completely right! The only problem here is that the scanner says that it *might* be a virus. If you think the scanner tells you it has found a virus, it turns out to be a false alarm. However, if you take this information as it is, saying 'OK, the facts you reported are true for this program; I can verify this so it is not a virus', I wouldn't count it as a false alarm. The scanner just tells the truth. The main problem here is the person who has to make decisions with the information supplied by the scanner. If they are a novice user, it is a problem. More about that later.

3.2 AVOIDING FALSE POSITIVES

The number of false positives can be minimized by the scanner author in several ways. The four techniques most often used are:

1) Definition of (combinations of) suspicious abilities

The scanner does not issue an alarm unless at least two separate suspected program abilities have been found.

2) Recognition of common program codes

Some known compiler codes or run time compression or decryption routines can cause false alarms. These specific compression or decryption codes can be recognized by the scanner to avoid false alarms.

3) Recognition of specific programs

Some programs which normally cause a problem (like the LoadHi program used in the example) can be recognized by the heuristic scanner.

4) Assumption that the machine is initially not infected

Some heuristic scanners have a 'learn' mode, i.e. they are able to learn that a file causing a false alarm is not a virus.

3.3 DEALING WITH FALSE POSITIVES

Some false positives are not easily avoided. So, the user has to deal with a certain number of false alarms and must make the final decision as to whether a file is infected or not.

For this reason it is important for the scanner to explain to the user the reasons why the program is suspect. If the scanner says: "this program might contain a virus", it isn't helping the user very much. However, if the scanner says that the program is able to remain resident in memory and able to format a disk, the user can more easily figure out what is going on. If a word processor gives such an alarm, it is extremely likely that the program carries a virus because word processors generally are not able to format disks and remain resident in memory. However, if the suspected file is a resident disk-formatting utility, then all of the suspected abilities can be explained by the intended purpose of the program.

Reason for suspicion: memory resident and disk-formatting abilities.

PROGRAM	PROBABLY
Resident disk formatter	No Virus (innocent)
Word processor	Malicious (virus)

Both programs cause the same heuristic alarms but the final conclusion is different.

Naturally, it requires an advanced user to draw a conclusion for the question "infected or not?". However, my opinion is that judging the results of any scanner (also conventional scanners) is a task for an advanced user only. If the scanner has a 'learn' mode, i.e. is able to remember which programs cause a false alarm, the initial scan should be performed by an advanced user but the subsequent scans (when the possible false positives have been eliminated) can be performed by a novice user. This is already common practice in most organizations.

Anyway, it isn't as bad as it seems as all other detection methods (including signature scanning) are known to cause some false alarms as well. Heuristics however has the advantage that it is able to supply you with enough information to establish for yourself whether a suspected file is a virus or not.

4 HOW DOES HEURISTIC SCANNING PERFORM?

Heuristics is a relatively new technique and still under development. It is however gaining importance rapidly. This is not surprising as heuristic scanners are able to detect over 90% of the viruses without using any pre-defined information like signatures or checksum values. The number of false positives depends on the scanner but a figure as low as 0.1% can be easily reached.

TbScan 6.02 used on the large virus collection of Vesselin Bontchev showed the following results:

SCANNING METHOD	7210 FILES	DETECTION PERCENTAGE
Conventional	7056	97.86%
Heuristics	6465	89.67%

A false positive test, however, is more difficult to perform so there are no independent results available.

5 COMBINATION OF CONVENTIONAL AND HEURISTIC SCANNING

Some people think heuristic scanning is a replacement for conventional scanning. In my opinion it is not. Heuristic scanning serves a very useful purpose when used in combination with conventional scanning. The results of both scanning methods can be validated by each other, thereby reducing false positives and also false negatives.

Combined result of analysis:

HEURISTIC	CONVENTIONAL	PROBABILITY
clean	clean	very probably clean
clean	virus	might be a false positive
virus	clean	might be a false negative
virus	virus	very probably infected
fn: 10%	fn: 1%	combined false negatives: 0.1%
fp: 0.1%	fp: 0.001%	combined false positives: 0.00001%

The chances of both the heuristic scanner and the conventional scanner failing is minimal. If both scanning methods have the same results, the result is almost certain. In the few cases that the results don't agree with each other, additional analysis is required.

TbScan 6.02 used on the large virus collection of Vesselin Bontchev showed the following results:

SCANNING METHOD	7210 FILES	DETECTION PERCENTAGE
Conventional	7056	97.86%
Heuristics	6465	89.67%
Combined	7194	99.78%

6 WHAT CAN BE EXPECTED FROM IT IN THE FUTURE?

➤ THE DEVELOPMENT CONTINUES

Most anti-virus developers still do not supply a ready-to-use heuristic analyser. Those who have heuristics already available are still improving it. It is, however, unlikely that the detection rate will ever reach 100% without a certain number of false positives. On the other hand it is equally unlikely that the amount of false positives will ever reach 0%.

➤ REACTION OF VIRUS WRITERS

An important issue is the effect on virus writers. It is likely that they will try to avoid detection by heuristic scanners. Until now the goal was to avoid detection by signature scanners and this was very easy to do, as it was sufficient to modify only a small part of an existing virus. Teenagers with some basic understanding of programming could do so easily . Avoiding heuristic scanners, however, requires a lot more knowledge, if at all possible.

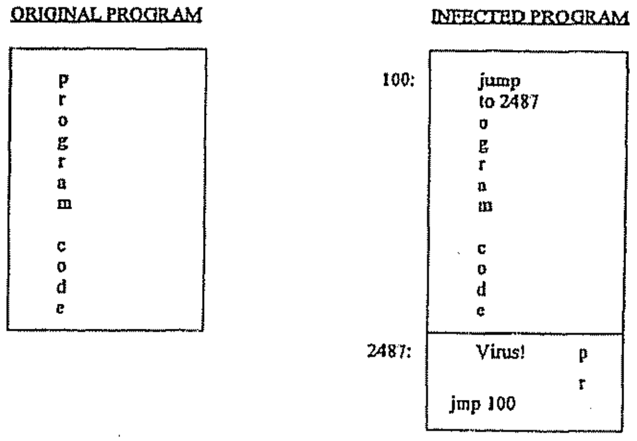
Fortunately, this detection-avoiding method of programming makes detection by conventional anti-virus products easier because it means that the programmer cannot use very tight and straight code. The virus writer will be forced to write more complex viruses.

7 THE PROS AND CONS OF HEURISTIC SCANNING

ADVANTAGES	'Future' viruses can be detected
	The user is less dependent on product updates
DISADVANTAGES	False positives are possible
	Judgement of the result requires some basic knowledge

8 HEURISTIC CLEANING

Before we can discuss heuristic cleaning, it is important to know how a virus infects a program. The basic principle is not difficult. A virus - a program by itself - adds itself to the end of the program. The size of the progmm increases due to this addition of the viral code. Appending a virus program to another program is however not enough, the virus code must also be executed. To make this happen, the virus overwrites the first bytes of the file with a 'jump' instruction, which makes the processor jump to the viral code. The virus now gains control when the program is invoked and it will later pass control to the original program. Since the first bytes of the file are overwritten by the jump instruction, the virus has to 'repair' these bytes first. After that the virus just jumps to the beginning of the original program and, most often, this program works as usual.



To clean an infected program, it is of vital importance to restore the bytes being overwritten by the jump to the virus code. The virus has to restore these bytes also, so somewhere in this virus code these original bytes are stored. The cleaner searches for those bytes, puts them back in their original location and truncates the file to the original size.

8.1 HOW DOES A CONVENTIONAL CLEANER WORK?

A conventional cleaner has to know which virus to remove. Suppose your system is infected with the Jerusalem/PLO virus. You invoke your cleaner and it proceeds like this:

"Hey, this file is infected with the Jerusalem/PLO virus. OK, this virus is 1873 bytes in size, and it overwrites the first three bytes of the original program with a jump to itself. The original bytes are located at offset 483 in the viral code. So, I have to take those bytes, copy them to the beginning of the file and I have to remove 1873 bytes of the file. That's it!"

8.2 SHORTCOMINGS OF CONVENTIONAL CLEANERS

The cleaner has to know the virus it has to remove; it is impossible to remove an unknown virus.

The virus should be the same as the virus known to the cleaner. Imagine what would happen if the virus used in the example was modified and now 1869 bytes in size instead of 1873... the cleaner would remove too much! This is not an exception but happens quite often since there are so many mutants. For instance, the Jerusalem/PLO family now contains more than 100 mutants!

Many polymorphic viruses have variable lengths and maintain the original instructions encrypted. Most conventional cleaners are therefore unable to clean MTE infected programs.

8.3 THE VIRUS WILL REMOVE ITSELF BEFORE ACTUAL EXECUTION

We have seen above how a virus works. The interesting part is that when the virus passes control to the original program it restores the original bytes at the beginning of the program and jumps back to start the program. Every virus is able to repair the original program in order to keep it functional (except for overwriting viruses, but these can't be cleaned anyway).

8.4 LET THE VIRUS DO THE DIRTY WORK

The idea is: why not let the virus do the dirty work? The basic principle of heuristic cleaning is simple. The heuristic cleaner loads the infected file and starts emulating the program code. It uses a combination of disassembly, emulation and sometimes execution to trace the flow of the virus and to emulate what the virus is normally doing. When the virus restores the original instructions and jumps back to the original program code, the cleaner stops the emulation process and says 'thank you' to the virus for its cooperation in restoring the original bytes. The, now repaired, start of the program is copied back to the program file on disk and the part of the program that gained 'execution' will be removed. An additional analysis of the cleaned program file will be performed to be on the safe side.

Note that the cleaner is actually removing the unknown from the unknown. No pre-defined information about the virus or infected file is necessary.

The process of emulation is just like hitchhiking. The emulator convinces the viral code that it is actually executing and it hitchhikes to the point where the virus passes control to the original program.

However, the actual process is very complicated. As with hitchhiking, many things can go wrong:

- ➤ The driver takes you to the wrong place

  The virus does not intend to execute the original program but it starts doing completely different things. As the purpose of the emulation is to restore the originalprogram, we never reach our goal.

- ➤ The driver won't let you out

  If the viral code performs an endless loop, the original program will never be restored so the cleaner might wait forever.

- ➤ The driver leaves the car

  A potentially dangerous situation is that the cleaner is too ambitious in its task to emulate everything and the virus gets control inside the emulated environment and finally escapes from it.

- ➤ The driver hits a tree and kills you too

  Many viruses are badly programmed. If they crash inside the emulator, chances are that the emulator crashes too.

Heuristic cleaners are so complicated that there is only one available right now. However, the great potential of heuristic cleaning make it likely that there will be more heuristic cleaners soon.

8.5 THE PROS AND CONS (OF HITCHHIKING)

ADVANTAGES	No need to recognize mutants No problems with polymorphic viruses 'Future' viruses can be cleaned User is less dependant on product updates
DISADVANTAGES	No exact copy of the original It cleans everything: even clean files!

Being the author of the first heuristic cleaner I have received many reactions to it. Most people were surprised that my cleaner was able to remove MTE viruses before my scanner was even able to recognize them. This is especially interesting as most anti-virus products are still not able to remove MTE infections.

Of course everybody wants to know how many viruses can be removed this way. I cannot give a reliable figure as testing a cleaner is an extremely tedious and time consuming task. However, a figure of 80% is a rough estimate. Many conventional cleaners do not even come close to this percentage.

8.6 WHAT CAN BE EXPECTED FROM IT IN THE FUTURE?

Heuristic cleaning needs additional improvements. Some viruses use anti-debugger features that also make an emulator fail. It is also still possible that a virus detects that it is being emulated, and it can simply refuse to cooperate. The better the emulator performs, the less likely this is. Major improvements, however, are more likely to show up when several more heuristic cleaners are available and some competition occurs.

EXHIBIT 11

MCF: A Malicious Code Filter[*]

Raymond W. Lo[**] *<lo@cs.ucdavis.edu>*
Karl N. Levitt <levitt@cs.ucdavis.edu>
Ronald A. Olsson <olsson@cs.ucdavis.edu>

Department of Computer Science
University of California, Davis
Davis, CA 95616-8562
Phone: (916) 752-7004
Fax: (916) 752-4767

[Address editorial correspondence regarding this paper to Olsson.]

May 4, 1994

[*] This work is supported by the United States Department of Defense, Lawrence Livermore National Laboratory, and Deloitte Touche.

[**] Current address: Raymond W. Lo, Silicon Graphics, Inc., 2011 N. Shoreline Blvd., P.O. Box 7311, M/S 10U-178, Mountain View, CA 94039-7311.

MCF: A Malicious Code Filter

Abstract

The goal of this research is to develop a method to detect malicious code (e.g., computer viruses, worms, Trojan-horses, and time/logic bombs) and security-related vulnerabilities in system programs. The Malicious Code Filter (MCF) is a programmable static analysis tool developed for this purpose. It allows the examination of a program before installation, thereby avoiding damage a malicious program might inflict. This paper summarized our work over the last few years that lead us to develop MCF.

- We investigated and classified malicious code. Based on this analysis, we developed a novel approach to distinguish malicious code from benign programs. Our approach is based on the use of *tell-tale signs*. A tell-tale sign is a program property that allows us to determine whether or not a program is malicious without requiring a programmer to provide a formal specification.
- We generalized program slicing to reason about tell-tale malicious properties. Program slicing produces a *bona-fide program*—a subset of the original program behaving exactly the same with respect to the realization of a specified property. By combining the tell-tale sign approach with program slicing, we can examine a small subset of a large program to conclude whether or not the program is malicious
- We demonstrated the capabilities of the tell-tale sign approach and program slicing to detect some common UNIX vulnerabilities.
- We determined how our basic approach can be defeated and developed a countermeasure—the *well-behavedness check*. Static analysis produces inaccurate slices on a program that has pointer overflows, out-of-bounds array accesses, or self-modifying code. The well-behavedness check applies flow analysis (integer-range analysis) and verification techniques (loop invariant generation, verification condition generation, and theorem proving) to identify such problematic cases.

Keywords

malicious code, malicious code detection, static analysis tool, program slicing

1. Introduction

Malicious programs can cause loss of confidentiality and integrity, or cause denial of resources. Common classes of malicious programs include computer viruses [4], computer worms [13], Trojan horses, and programs that exploit security holes, covert channels, and administrative flaws to achieve malicious purposes.

Some program properties allow us to discern malicious programs from benign programs easily with very high accuracy without the need to give a specification of the program. We call these properties *tell-tale* signs. The idea is to program a *filter* to identify these tell-tale signs. Nevertheless, the filter may mistakenly identify some normal programs as malicious (false positive). The goal is to minimize such mistakes.

For example, we can use tell-tale signs to identify computer viruses. Consider a hypothetical virus W that infects writable and executable programs. W infects a program if the file has enough empty space at the end of the program by 1) copying its viral code to the end of the text segment; 2) modifying the entry point of the victim program to viral code; and 3) registering the original entry point so that control is passed back to the original program when the virus finishes executing. We may identify a program infected by the following tell-tale signs.

- Duplicated system calls. The original program has one open() system call. Since the viral code carries its own open() system call, the infected program has two open() system calls.
- Isolated/Independent code. A viral code is typically self-contained and independent of the infected program. No shared (global) variables and parameters are passed between the viral procedure and other program procedures.
- Access of text segment as data. When the virus copies itself to other programs, it reads the viral code from its own text segment. Reading the text segment is a rare activity in normal programs.
- Anomalous file accesses. The virus opens and writes to executable files, normally only done by compilers and linkers.

These tell-tale signs do not identify only the W virus, but also others. For example, we can detect the RUSH HOUR virus [3], which was developed and published for virus demonstration, using the fourth tell-tale sign. The RUSH HOUR virus is intended to harmlessly show the danger of viruses to computer systems. The virus only lodges itself in the MS-DOS German keyboard driver KEYBGR.COM. When the virus is in the system, it searches the current directory for the keyboard driver every time the user accesses the disk. The virus, which camouflages itself as a keyboard driver, intercepts all MS-DOS system calls: The infecting action is triggered by the load-and-execute system call. After being triggered, the virus tests the KEYBGR.COM on the specified drive and infects it, if it has not already been infected. We use our tool to look for file access system calls in MS-DOS system files and device drivers, which should not have any.

As another example, a time-bomb can be easily detected using the tell-tale sign approach. A time-bomb contains malicious code that is triggered at a certain time. A generic time bomb, as shown in Fig. 1, first reads the current time, and then compares it with a triggering condition. If the triggering condition is satisfied, the time bomb performs the damages. The security analyst can program MCF to recognize such an execution pattern (the time-dependent execution of certain statements) that is rather suspicious.[1]

[1] There are just a few exceptions, e.g., the UNIX make program, incremental backup procedures, or editors like emacs.

```
time-bomb:
  now = gettimeofday();
  if (trigger-time(now))
    do-damage;
  ...
```

Fig. 1. Program skeleton of time bombs

The tell-tale sign approach can detect unseen, but similarly structured malicious code. If new malicious code undetectable by existing tell-tale signs is found, MCF can handle new tell-tale signs for detecting the new malicious code. Since MCF uses static analysis to consider all possible execution paths of a program, it can identify problems not detected using run-time or dynamic analyses. By combining the tell-tale sign approach with program slicing, we can just examine a small portion (i.e., the security related portion) of a program to conclude whether or not the program is malicious; for programs with hundreds or thousands lines of code, these slices are often just a few lines. Compared with other static analysis techniques that must examine the whole program, we believe our approach imposes the minimal amount of work required by using program slicing. With the use of well-behavedness check, we can identify situations that a static analysis tool might be fooled by a malicious code. Existing tools do not identify such cases and thus cannot provide a level of confidence comparable to our tool.

Section 2 compares other malicious code detection approaches with ours. Section 3 contains more tell-tale signs that can detect other classes of malicious code and system vulnerabilities. Section 4 gives examples in applying these tell-tale signs. Section 5 describes applying program slicing to mechanize the identification of tell-tale signs. Section 6 contains the analysis of one user program and one system program. Section 7 describes how MCF can be defeated and introduces the well-behaved property. Section 9 concludes the paper. This paper summarizes our approach; complete details appear in [10].

2. Related Work

The simplest approach to detect malicious code is to run the program to see whether it shows any viral activities. Despite its simplicity, run-time approaches have several major drawbacks. First, they expose a system to potential damages by running a potential malicious program. Second, they only detect and then inhibit malicious programs' activities, but they cannot identify the presence of malicious code when the code is dormant. Third, when a run-time tool identifies a problem, it either stops the malicious program or asks for human attention. For systems running without attention, run-time approaches are simply not viable.

Static approaches perform the analysis without executing the program. Therefore, they do not have the problems associated with run-time approaches. However, static analysis is harder to implement. Current static methods are comparison-based. They fall into the following three general categories according to whether the program is 1) compared with a "clean" copy of the program [5], 2) compared with known malicious code (used by virus scanners), or 3) compared against a formal specification [7]. Unfortunately, a "clean" program is not easily obtained, the most dangerous malicious code are the unknown ones. Also the formal specification and verification of programs is at best difficult. Commonly used programs often have no specifications and are very unlikely to be verified.

Dynamic analysis [6] combines the concept of testing and debugging to detect malicious activities by running a program in a clean-room environment. The execution is typically monitored (e.g., by a programmable debugger [11]) for suspicious behavior. The analysis is in general more reliable than run-time approaches because data are generated systematically to test the program [9]. Test coverage analysis will also reveal parts of programs not covered by the analysis. Compared with static analysis, dynamic analysis is less reliable because testing can never be exhaustive.

Malicious code can be detected by a human analyst screening the program. Although a human can reason about a program in detail, s/he is weak in examining code and data that are spatially or temporally separated, and also has difficulties in handling large amount of information at one time.

A malicious program may exploit the human weaknesses by obfuscated programming techniques such as using macros, overflowing pointers, writing self-modifying programs, or installing sections of malicious code in spatially separated parts of the program. Furthermore, a malicious code may use familiar variable names and procedure names associated with benign purposes to camouflage the malicious code. Finally, humans err. Thus the result of analysis by humans is not reliable.

Virus scanners are the only automated tool available nowadays for malicious code detection. They detect known viruses by scanning binary programs for pre-determined machine code sequences. The idea of scanning known malicious code is not very useful for detecting general malicious code because identical time-bombs or Trojan horses are unlikely to be found in different sites. Virus scanners are also not effective against polymorphic viruses.

3. Tell-Tale Signs

As mentioned in Section 1, tell-tale signs are properties of programs that can be used to discriminate malicious and benign program. Tell-tale signs must be simple enough so that their identification can be mechanized and must be fundamental enough so that certain malicious action is impossible without showing tell-tale signs. Most tell-tale signs are related to system calls because these system calls are the only way of performing certain functions. The following are some of the useful tell-tale signs. We use program slicing to reason about tell-tale signs. The program slices with respect to the tell-tale properties are usually short. Interestingly, many slices corresponding to the tell-tale signs are just empty, and very often a slice corresponds to more than one tell-tale signs. The work required by the analyst is, in fact, much less than it might appear. We believe that by examining these signs we can identify most malicious code. For convenience, we group the tell-tale signs into three groups.

3.1. Tell-tale Signs Identified by Program Slicing

These tell-tale signs apply to all kinds of programs and are used with the program slicer.

- **File Read.** This includes the slicing for the open() system calls. The list of files being read will show what kind of information the program may access (e.g., strange accesses to /dev/* should be detected).
- **File Write.** In addition to the open() system call, it includes the uses of create(), link(), and unlink() system calls because a file modification can be simulated by deleting and creating a file. The files written to should be checked against a list of important system files (e.g., /vmunix, /etc/passwd, /etc/aliases, /bin/*, /usr/bin/* files).[2]
- **Process Creation.** A malicious program uses the fork() system call to create processes. A denial-of-service malicious program may put a fork() system call in a loop to create a large number of processes.[3]
- **Program Execution.** A malicious program may create another process to perform the malicious action, so we check which other programs are invoked and examine them. Typical sequences are a fork() system call followed by an exec() system call, and the system() and popen() library calls.
- **Network Accesses.** Malicious programs can use the network to send information back to the writer. We will slice for the network system calls, e.g., socket(), connect(), send().
- **Change of Protection State.** We slice for the change of protection-states system calls, e.g., chmod() and chown(). It is rather unusual for normal programs to use these system calls and could indicate the presence of a trojan horse.
- **Change of Privilege.** We slice for the setuid() and setgid() system calls.
- **Time-Dependent Computation.** We find out how the time is used in the program. A forward slicing on the gettimeofday() system call shows all variables that contain time-dependent values. We will slice again for the

[2] Symbolic links to these files could exist. We depend on intrusion-detection systems to notify the system administrator when such links are made.

[3] The number of processes created is limited by the maximum number of processes per user in some UNIX systems.

statements depending on some time-dependent values.

- **Input-Dependent System Call.** This tell-tale sign refines the file open tell-tale sign. Some UNIX applications have data-flow paths from a read() system call to a open() system call. That means a user can probably control which files these applications can modify by supplying certain inputs.
- **Race Conditions.** Race-condition bugs occurred in some root-privileged UNIX system utility, e.g., rdist and fingerd. In both cases, the requested file/directory accesses are validated before files are opened. An intruder may relink the file/directory in the period between the validation and the actual access. This situation can be characterized by an access() system call preceding an open() system call.

3.2. Tell-Tale based on Data-Flow Information

These signs include anomalous pointer aliasing, data dependence, anomalous interprocedural data dependence. They do not need the program slicer.

- **Anomalous Data Flow.** This relates to possible bugs in a program. Some detectable anomalies include consequent definition of variables without any usage in a path, use of undefined variables, and branch testing that depends on a constant value.
- **Anomalous Interprocedural Data Dependence.** We compute the summary data-flow information for each procedure and create a data-dependence graph in which a node represents a procedure and an edge represent data-flow. Malicious code (e.g., viruses) that does not use any value computed in the original program will show up as a disconnected component in the summarized data-dependence graph.
- **Well-Behavedness.** Bad-behaved programs can fool static analysis tools. Two checks are required: 1) that dereferenced pointers contain valid addresses and 2) that pointers/arrays do not overflow. We also look for uses of the gets() library call that do not limit the size of the input string, such as the well known finger daemon bug (see Section 4.2.1). Details are in Section 7.

3.3. Program-Specific Tell-tale signs

The above tell-tale signs apply without needing to know what the program does. If we can determine the function of the program, more analysis can be done. The tell-tale signs in this section include properties of system programs we should examine. These properties are complicated and typically require significant human analysis, but with the use of program slicer, the effort is drastically reduced. Furthermore, the security analyst can examine additional properties pertinent to certain classes of programs.

- **Authentication.** We want to find out how authentication is performed. We slice for the conditions that are true for the authentication to be granted.
- **Identification of Changes.** This detects what information is changed. For example, the telnet program should pass information back and forth without modification. The chfn (change finger name) program should only modify the database information field of the password entry. We can slice the program between the corresponding read() and write() system call for the modification of the values.
- **Internal State of Authentication Loop.** An authentication loop should be stateless. Its outcome should only depend on the userid, the password, and the password file; it should not depend on any global or static variables. The state of a loop can be derived easily using the data-flow information. This tell-tale sign has been used to identify a bug in ftp that caused a security problem.

4. Detecting Malicious Code and Common Vulnerabilities

4.1. Detection Malicious Code

The following malicious code are described according to the six steps of the malicious code model mentioned in Appendix B. The six steps are: 1) gain access to the system, 2) obtain privilege, 3) wait for triggering conditions, 4) perform malicious action, 5) clean up, and 6) repeat steps 1 through 5.

We have included a Trojan login program and multistage malicious code here. More examples including a salami attack program, a sniffer, a ferret program, and a program that overloads a system can be found in [10]. Although these programs are not real malicious code, they are based on realistic examples and are used to illustrate how tell-tale signs are useful towards detecting real malicious code.

4.1.1. Trojan Login

The Trojan login program is usually advertised as some enhancement to the existing login program (e.g., to use shadow passwords) and works as follows:

(1) It is copied to the system by the administrator.

(2) It is installed in the /bin directory as a root-setuid program.

(3) An outsider enters the system using a bogus userid, for which a password is not required by the Trojan login program.

(4) A root-privileged shell is created for that particular login.

(5) The login program does not write the bogus login to the log file, so the bogus login will not show up with system-administration programs (although it could show up in a command-log file).

The Trojan horse code is detectable with the "Authentication" tell-tale sign. There is a path starting from the entry point to the privilege-granting part without password checking. The analyst will need to locate the privilege-granting setuid() system call and then slice for the authentication code. With the Trojan horse, the analyst should identify a path to the setuid() system call that does not pass through the password-comparison code.

4.1.2. Multistage Launcher

This mechanism carries a malicious program into a specified location (system). The mechanism is similar to how that used by viruses to replicate, but the malicious program replicates in a controlled way and has a target. The program has no specific malicious action except propagating to more secure systems. The triggering and the action of the malicious code is programmable. For example, it can be programmed to deliver other malicious code (such as the malicious code described in the next section) into a development system as follows:

(1) The multistage malicious program is installed as a Trojan 'ls' program in the /tmp directory by an insider.

(2) Users working in the /tmp directory may execute the Trojan 'ls' program accidentally.

(3) After invocation, the malicious program determines whether it should migrate to a remote system accessible by the current victim (i.e., whether the remote system is closer to the target machine).

(4) If the malicious program migrates, it copies itself to the file /tmp/ls on the remote machine.

(5) The program avoids detection by maintaining one copy of itself all the time.

(6) The program repeats Steps 1 through 5 until the specified machine is reached.

The multistage program executes rcp or rsh to transfer itself from one machine to another. The execution of rcp or rsh is discovered by the "Program Execution" sign.

4.1.3. Development System Attack

This attack is aimed at embedded system. The malicious program in this attack has two stages. The first stage gets into the development system and installs the second stage in the weapon system. The second stage malicious code creates a blind spot in the firing-control component in an embedded weapon system. An example of such an attack is as follows:

(1) It uses the multistage launcher to get into a development system.

(2) It is executed by a system administrator.

(3) The action is triggered when the program has the privilege to modify the library file (e.g., the C library /usr/lib/libc.a in UNIX).

(4) It changes the sin() function in the library, so that $sin(x) = sin(45)$ when $44 < x < 45$. The effect is that the firing system (the gun activator) can never aim at an angle between 44 and 45, thus it provides the enemy with a safe direction of attack.

(5) The program eliminates itself once the library is modified.

The development-system attack program is carried by the multistage launcher. Since this program damages a system by modifying its functionality slightly, there is no effective way to identify it (because there are so many ways to change a functionality and there are so many functionalities in a system). However, it is still detectable because we can detect the launching section as mentioned above and the modification of the library with the "File Write" sign.

4.2. UNIX Vulnerabilities and Their Detection

In this section, we examine how the tell-tale sign approach is useful for identifying some known system vulnerabilities. More examples including the rdist bug and the sendmail bug can be found in [10].

4.2.1. Finger Daemon (fingerd)

The finger daemon (fingerd) has a bug that allows an intruder to read protected files without proper privilege. fingerd, running as root, prints the content of the .plan file of the person being fingered. Therefore, an intruder can symbolic-link his .plan file with a protected file and then run finger, which invokes fingerd, to print out the content of the protected file. This bug was fixed by first checking that .plan is not a symbolic link before opening the file. However, this fix can be circumvented if the intruder links the .plan file during the period after the check has finished and before the open() system call executes. This race condition is detectable by the "Race Condition" sign.

4.2.2. Mail Notifier (comsat)

The utmp file records information about who is currently using the system. Whenever a user logs in, login fills in the entry in /etc/utmp for the terminal on which the user logged in. /etc/utmp is owned by root but is world writable. Anyone who has an account on the system may modify /etc/utmp. If the system enables tftp, /etc/utmp can be modified from other systems.

The mail notifier (comsat) is the server process that waits for reports of incoming mail and notifies users who have requested to be told when mail arrives. Comsat listens on a datagram port associated with the biff service specification (see services in Section 5 of Unix man pages) for one-line messages of the form user@mailbox-offset. If the user specified is logged onto the system and biff services have been turned on, the first part (10 lines) of the mail is printed on the user's terminal. Comsat reads the file /etc/utmp to determine the appropriate terminal to which to write the mail message. Furthermore, comsat is run as root.

An intruder can modify the terminal field in his /etc/utmp entry to /tmp/x and link it to a system file, e.g., /etc/passwd. Then he can turn on the mail-notification service and send himself mail. Comsat will write the first few lines of the mail message to the target file. If the target file is the password file, the hacker can supply a bogus password entry in the mail he sent himself.

The comsat problem is revealed by the "File Read" sign, which indicates that the file written to comes from the /etc/utmp directly. Further analysis on the access of /etc/utmp shows that its content is not validated.

5. Mechanizing Malicious Code Detection

Program slicing [16] produces a bona-fide program—a subset of the original program that behaves exactly the same with respect to the computation of a designated property. The concept of breaking down a large program into smaller modules for analysis dates back to 1975 [17]. Zislis uses busy variables (variables that will be used later in the program) as the criteria to group related program statements together and form a slice. Weiser [16] uses a more accurate criteria—data dependence—to group statements together. These criteria are not the only ways of grouping relevant and eliminating irrelevant statements. In this section, we discuss several ways of applying the control-dependence and data-dependence analyses to "slice" a program—namely, backward data-flow slicing (Weiser-style slicing), forward data-flow slicing, predicate-region slicing, and control-flow slicing. These ways are used to identify different tell-tale signs but they employ the same platform for analysis.

5.1. Program Representation

The program being analyzed is translated into an intermediate form. We represent the intermediate form with a program graph. For convenience of analysis, we impose the following restrictions (some achieved through program transformation) on the intermediate form:

- a branch node is split into a true-branch and false-branch node to distinguish their influences.
- expressions have no side effects, but procedures can.
- at most one procedure call is allowed in each computation node.
- at most one variable is modified in each computation node.
- the data-flow definition of all system and library calls are pre-determined.
- all storage locations are identified and given a name. We call them *objects*.
- all pointer variables must point to some objects or have the value NULL.

5.2. Global Flow Analysis

Most compilers perform only intra-procedural analysis because of the limited time allowed to be spent by the optimizer. It is safe to make certain assumptions, e.g., that local variables are not modified by other procedures. In security analysis, the analysis must be global, inter-procedural, and have the assumptions validated. Malicious code writers will not conform to rules of good programming practice to make our lives easier, e.g., a procedure in a malicious program may interfere with other procedures through legitimate (aliasing) and non-legitimate means (pointer overflows).

We perform a global pointer mapping analysis to determine the effect of pointer aliasing on the data dependence by keeping track of the values of each pointer variable. Then we compute the data and control dependence for the entire intermediate program. We provide the following functions after completing flow analysis.

- *pred*(u) returns the set of nodes that can reach u.
- *succ*(u) returns the set of nodes that u reach.
- *forward-depend*(u) returns the set of nodes in which the computation uses the value of a variable modified in u.
- *backward-depend*(u) returns the set of nodes that modifies a variable used in u.
- *predicate-depend*(u) returns the branch nodes that decide whether or not u executes.
- *predicate-region*(b) returns the set of nodes that is executed if the branch node b is taken.

5.3. Program Slicing

We perform slicing on a per node basis. A program slice is represented by a set of nodes. Given the set of nodes and the original intermediate program, a subset program can be reconstructed easily. Since a program slice is represented by a set, it is possible to combine the effect of different slicing methods by set-union, set-intersection, or re-slicing using different criteria. In the following discussion, *focus* is used to combine different slicing methods. Notice that *focus* initially contains the whole program.

Control-flow slicing is extremely simple. Since there is no reason to look at complete execution paths all the time, we can eliminate those sections in which we are not interested. For example, when slicing for the file accesses call, we are interested in sections of paths starting at the entry point and ending at an open() system call. For programs including authentications, we may only be interested in the authentication section.

The control-flow slicer accepts two points—u and v—in a program and determines the nodes in any path going from u to v. The slicing is produced by the following equation:

$$\textit{control-slice}(u, v) = \textit{focus} \cap \textit{succ}(u) \cap \textit{pred}(v)$$

Weiser uses backward data-flow slicing. Informally, it determines which statements affect the variables at the statement under examination. A statement can affect a subsequent statement either directly or indirectly. The *direct* effect provides a value to be used at the later statement. The *indirect* effect controls whether the later statement will be executed. In Fig. 2, statement 3 has a direct effect on 4 because y:4 (y:4 represents the value of y at line 4) uses the value x:3; statement 2 has an indirect effect on statements 3, 4, and 6 because it determines which of them are executed.

```
c = 1;
if (c) {
   x = 10;
   y = x;
} else
   y = 3;
```

Fig. 2. Direct and indirect data dependence

The slicing algorithm, shown in Fig. 3, is a general slicer that can produce a program slice by collecting direct, indirect, or their combined data-dependence in a forward or backward manner. The variable *focus* carries the part of the program narrowed down by previous slicings.

```
general-slice(nodes, focus, direction, dependence)
{
  new-list = {nodes};
  node-list = { };
  while (node-list ≠ new-list) {
    node-list = new-list;
    if direction is forward {
       if dependence is "indirect" or "both"
          new-list = forward-depend(node-list);
       if dependence is "direct" or "both"
          new-list = predicate-region(new-list);
    } else if direction is backward {
       if dependence is "control" or "both"
          new-list = backward-depend(node-list);
       if dependence is "data" or "both"
          new-list = predicate-depend(new-list);
       }
    new-list = new-list∩focus;
  }
  return node-list;
}
```

Fig. 3. General program slicer

Backward data-flow (Weiser's) slicing determines the set of statements that affect the variables directly or indirectly at the statement under examination. It is defined as follows:

backward-both-slice(*node*, *focus*) = *general-slice*(*node*, *focus*, "backward", "both")

Forward data-flow slicing determines the effect of certain computations such as in the program. It is very similar to backward data-flow slicing, except that it traces forwards through data-flow graph and predicate-regions. It is defined as follows:

forward-both-slice(*node*, *focus*) = *general-slice*(*node*, *focus*, "forward", "both")

5.3.1. Slicing for File Access

Forward or backward slicing sometimes generates program slices that have too much details. With the file access properties, we are interested in which files are opened and not interested in under what situation the files are opened. Therefore, the nodes included by tracing the indirect effects are often useless. As the first approximation, we slice for the direct effects only; that usually produces a smaller slice that is also simpler to be examined. It is defined as follows:

backward-direct-slice(*node*, *focus*) = *general-slice*(*node*, *focus*, "backward", "direct")

5.3.2. Slicing for Time-Dependent Computation

The time-bomb example in Section 1 requires a different kind of program slicing, in which the direct effects are collected first, and then indirect effects are identified. The time-bomb slicing algorithm can be built as follows:

```
timebomb-slice(node, focus) =
  general-slice(
    general-slice(node, focus, "forward", "direct"),
    focus,
    "forward", "indirect");
```

5.3.3. Slicing for Race Conditions

We perform this slicing for pairs of access() system call and open() system call. First we apply control slicing to focus on the program nodes between the access and open calls. Then we perform backward slicing to see whether their arguments have common ancestors. (We should also check that if both system calls have constant arguments, the constants are different.)

```
Let anode contain an access() system call.
Let onode contain an open() system call.

race-cond-slice(anode, onode) =
   general-slice(anode, afocus, "backward", "direct")
      ∩
   general-slice(onode, ofocus, "backward", "direct")
  where afocus = control-slice(entry, anode)
   and ofocus = control-slice(anode, onode) ∪ afocus
```

Notice that this slicing can discover careless programming mistakes, but not all intentional malicious code. For example, the arguments to the two system calls are computed using distinct sets of variables (hence their slices do not overlap) to the same value.

5.3.4. Slicing for Other Signs

To identify the slice for the "change of protection state" sign, backward data-flow slicing is applied at the chmod() and chgrp() system calls. The slices for other tell-tale signs are produced with their corresponding system calls in a similar way.

6. Malicious Code Detection Example

This section presents a few examples to demonstrate the use of tell-tale signs and program slicers. The first example is a user game program that has a time-bomb embedded. The second example is a system login program. The analysis of a user program is much easier since most slices corresponding to the tell-tale signs are just empty. The analysis of the login program is more complicated because we need to examine the authentication logic. Appendix C contains the programs' complete source code.

In summary, the analyst needs to examine less than 10 lines of the 317-line hangman.c program and less than 100 lines of the 595-line login.c program. We expect the percentage saving is huge for large user programs because the portion of a program relating to our tell-tale signs are relatively constant.

6.1. Analysis of a malicious hangman program

The game program hangman.c is very simple in terms of slicing for any security-related properties because it writes no file; creates no process; and does not access the network, change protection states, change privilege, have input-dependent system calls, or contain any authentication code.

hangman.c reads only one file: /usr/dict/words.

```
302     if ((Dict = fopen("/usr/dict/words", "r")) == 0) {
```

The hangman program has the following call structure:

Caller	Callees
main	setup playgame
getword	abs
endgame	prman prword prdata readch
getguess	readch
playgame	getword prword prdata prman getguess endgame

We further summarize the data used and generated by each procedure. No independent computation is found—data is passed as parameters, return values, and also as global variables.

hangman.c uses the current time as the seed for the random number generator. The current time is obtained at statement 301 and used by srand(). After relating the flow with the static variable shared by the libraries srand() and rand(), we see that the time is used by fseek(). Furthermore, we see the value of time is compared with a constant at line 309 and stored in the variable Count. Then the statement 112 (i.e., a simulated time-bomb) is executed dependent on its value. So, the slice is:

```
308   srand(time(0) + getpid());
309   Count = (timeval >= 714332438); /* Aug 20 1992 10:45am*/
179   fseek(inf, abs(rand() % Dict_size), 0);
111   if (Count) /* Triggered after Aug 20 1992 10:45am */
112       printf("Time Bomb Triggered !!!\n"); /* Simulated Time-Bomb Action */
```

The manual detection of such a time-bomb would be difficult because of the spatial separation of the statement of comparing time (line 309) and the time-triggered action (lines 111 and 112), and that the name of the variable Count implies it does nothing related to the value of time. (Of course, someone reading hangman.c might notice the give-away comments and string on lines 111 and 112!)

Suppose the time-bomb is not embedded in this program. Then, the slice for "time-bomb" is:

```
308   srand(time(0) + getpid());
179   fseek(inf, abs(rand() % Dict_size), 0);
```

We see that no time-dependent computation is made and conclude the program is safe.

6.2. Analysis of login.c

We first locate the open() system calls, and then use approximate backward data-flow slicing to determine the value of the filename arguments. login has five open() system calls. /etc/nologin and /etc/motd are read. /etc/utmp, /usr/adm/wtmp, and /usr/adm/lastlog are modified. Our analysis proceeds as follows.

We find one execlp() system call; the program executed is stored in pwd->pw_shell.

Login has no direct network accesses.

Login uses chown() and chmod(), which in turn use ttyn and pwd as arguments. Login uses setuid(pwd->pw_uid) and setgid(pwd->pw_gid). They depend on the variable pwd.

We slice for time-dependent computations. We identify one time() library call, but no statements executed depending on the value of time. The time records the login time of a user.

We identify whether any input values affect some security-related system calls. We try to locate paths leading from a read() system call to a open() system call. No such paths are found.

The program has a very flat call structure. main() calls doremotelogin(), getloginname(), rotterm(), showmotd(), stypeof(), doremoteterm(), and setenv(). doremotelogin() calls getstr().

The program has three disconnected components by considering aggregated data flow at the procedural level, as shown in the following:

- main, doremotelogin, getloginname, rotterm, showmotd, stypeof, doremoteterm, getstr
- timedout
- catch

The first one is the main body of the login program. The other two are the signal handlers implementing timeouts. After examining timedout and catch, no malicious code is found.

We use control-flow slicing to narrow the search in the program between an access() and an open() system call. Then we use backward data-flow slicing for the arguments in the open() system call. Only one access() is found, and its argument qlog is not used by any open() system call. Therefore, login does not have this race condition.

We need to slice for the authentication code, that is determining under what situations setuid(), chown(), etc. are executed. To slice the authentication loop, we use control-flow slicing to focus on the program fragment before and in the loop, and then we slice for the conditions (i.e., slicing for invalid) that the loop may exit. In login, the loop exits mean that the authentication is accepted. About 100 lines of C statements are collected for analysis by the security analyst, who after carefully examining the code determines the program does what it should.

Statements 183 to 288 are the authentication loop—if the authentication fails, the program obtains another userid and password and retries. We try to determine the state variables of this loop. A variable is a state variable if it is also an induction variable (i.e., the current iteration depends on some values computed in previous iterations). The induction variables in the authentication loop are pwd, utmp, lusername, argc, and invalid. Although an authentication routine should not have state variables, careful examination of the loop shows that login is correct. Since the authentication (password checking) should be stateless (other than storing the userid), the authentication can be rewritten in a way to eliminate the induction on pwd, utmp, argc, invalid. The resulting program is much easier to understand and analyze.

7. Defeating MCF (Stealth Techniques[4])

We think that a good malicious code detection tool should disclose the ways that it might be compromised because a malicious code writer will surely learn of the existence of a detection tool and of its detection method. Once a method to defeat a tool is found, the method can be automated to convert existing malicious code to

[4] We name the techniques used by existing and future malicious code to avoid detection stealth techniques, following the naming of stealth viruses.

undetectable malicious code. For example, virus scanners are found to be useless against polymorphic viruses. A toolkit that converts existing PC viruses to polymorphic viruses has been developed and exchanged among virus writers [14]. Furthermore, the detection tool should also identify cases in which its result might be unreliable.

To fool our analysis tool, a devious programmer may use array/pointer overflow to confuse the data flow analyzer, or use array/pointer overflow to change the control flow of the program or to execute data. If the devious programmer uses array/pointer overflow to modify data flow to confuse data flow analyzer that the program slicer depends on, the modification is not represented in the data dependence graph. The devious programmer can use array/pointer overflow to modify the return address on a stack. The execution sequence of the program is different from what is perceived by the analyst or our analysis tool. The malicious program can execute data or self-modified code. Both our tool and the analyst examine program statements for malicious activities. The devious programmer can hide the malicious code by embedding them in the the data storage area, and then transferring control to the data. Examples of such programs are in Appendix A.

We can detect these stealth techniques by validating our assumptions about programs. These stealth techniques fail if the analyzed program satisfies the following requirements.

- The program does not modify its code.
- The program does not transfer control to data.
- The program does not allow modification of variables that have not been identified by the data flow analyzer.

These requirements are further translated into two properties: the well-formed and well-behavedness property. The well-formed property governs the generation of pointer values—all pointers must point to some variables or procedures, or have the null value, as mentioned in the program representation. The well-behavedness property states that there is no modification through overflowed arrays or pointers and no modification through procedure pointers. Therefore, all data dependence can be considered by the program slicer. If the two properties are satisfied, the program slice corresponds to the original program with respect to the slicing criteria. The function of the well-behavedness checker is to verify these properties.

We have developed a well-behaved checker that applies both flow analysis and verification techniques to show that pointers do not overflow and array-accesses are within bounds. Details can be found in [10] Nevertheless, the checker can verify most array accesses automatically, but there are some cases that the tool cannot handle.

8. Conclusion

Tell-tale signs are useful in discriminating malicious from benign programs. Since no discrimination method is perfect, as shown by Cohen [4], we identify a larger class of program called suspicious programs. Suspicious programs are those that carry code that *might* perform malicious action. Tell-tale signs can identify such programs. Selecting good tell-tale signs would reduce the cases that a program is found suspicious but not malicious (i.e., false positive), and minimize undetected malicious code (i.e., false negative). We conjecture that it is difficult to write malicious code that can bypass our small collection of tell-tale signs. If such malicious code can be written, we can easily update our library of tell-tale signs to detect it.

The use of program slicing to determine tell-tale properties reduces the work of the analyst when s/he has to examine a program. In the future, systems (using dynamic analysis and testing techniques) might be developed to examine these slices so that the detection process is more automated.

We made several major improvements over existing and proposed malicious-code detection methods. We do not require a formal specification of the program being analyzed. The tell-tale sign approach is general enough to identify classes of malicious code, whereas other approaches may handle only one instance of malicious code at a time. Our tool is programmable so that it can be adapted to handle new malicious code. Most important, previous work offering a similar level of confidence does not exist

The problem with our tool is that it does not work with self-modifying programs (but can detect them). The usefulness of our tool depends on how the program is written, i.e., the use of pointers, dynamic memory allocation, and recursive data structures increase the size of program slices. The correctness of its result relies on the verification of well-behavedness property, which unfortunately cannot be completely automated.

We foresee that programming languages will be designed with more concrete semantics and constructs that are easier to analyze. With high-assurance software, certain programming methodology and styles will be followed, leading to programs that are more sliceable and more easily analyzed.

In terms of the development of the Malicious Code Filter (MCF), we envision that MCF will be operated in two modes. In the first mode, MCF will act as a coarse filter, identifying those programs worthy of closer examination. MCF will analyze a program and summarize its properties to allow the analyst to understand the possible effects of its execution. In its second mode of operation, MCF will support a more detailed examination of a sliced program, perhaps one that has been identified as such by an earlier MCF. This analysis will investigate the exact nature of the previously identified suspicious property, determine its triggering conditions, and possibly discover additional suspicious properties. So far, MCF operates only in the first mode. Techniques such as symbolic evaluation [2], dynamic analysis [11,12], and testing [9] will be very useful in building the second mode.

Appendix A: Examples of Bad-behaved Programs

```
/*
  Stealth programming using pointer overflow:
    The pointer p is overflowed to point the string "siruv".
    By dereferencing p, we can actually change the string "siruv" to "virus".
    The data dependence graph shows nothing about the string modification.
*/
main()
{
    int i; char *p, c;
    p = "nothing" + 8;              /* the offset 8 is system dependent */
    c = *(p+4);  *(p+4) = *p;  *p = c;
    puts("siruv");
}
```

```
/*
  Stealth programming using control flow modification:
    The main procedure modifies its return address by overflowing
    the array x and replacing the return address in the stack with the
    address of unreachable(). unreachable() is executed
    when main() returns.
*/
unreachable() {
    puts("virus"); exit();
}
main() {
    int x[1];
    /* the offset of the return address from x, 2 * sizeof(int),
          is system dependent */
    x[2] = unreachable;
}
```

```
/*
  Stealth programming using data execution:
    This program executes on a Sun 3 workstation.
    data[] contains a machine code program to print out the string "virus".
*/
data[] = {
 0x4e560000, 0xdffc0000, 0x48d7, 0x4878, 0x6487a, 0x1c4878, 0x161ff, 0xc,
 0x4fef000c, 0x4e5e4e75, 0x48780004, 0x4e404e75, 0x76697275, 0x730a0000, 0
};
main() {
  int (*f)();
  f = (int (*)()) data;
  (*f)();
}
```

Appendix B: Malicious Code Model

Malicious code exhibits anomalous behavior, e.g., reading protected files, modifying protected files, and obtaining unauthorized privilege. Based on our investigation of the activities of malicious code, we express their anomalous activities as six steps in performing malicious actions.

(1) **Gain access to the system.** A malicious code must be installed in a system before it can be activated. It may be installed by an insider who has the appropriate privilege. As a Trojan horse, it may be installed by casual users who obtain the malicious code from a public bulletin board. As a virus, it may attach itself to a user's diskette when the user accesses an infected machine. To a lesser extent, an outsider who does not have direct access to the system can install malicious programs through known OS bugs or flaws [15] in protection settings (protection states).

(2) **Obtain higher privilege/Retain current privilege.** Once a program is installed in the system, it may belong to a particular user in the system, but it may not have sufficient privilege to perform the malicious action. The malicious program may want to retain the privilege beyond the termination of the current process, so that the malicious action can be performed at a later time.

There are many ways to expand the privilege in a UNIX system. As mentioned in Step 1, the malicious code can exploit bugs in OS and privileged applications, or incorrect protection settings. The protection settings of a UNIX system can be legitimately altered directly or indirectly. With the direct methods, the file access mode, setuid bit, setgid bit, the file owner id, and the group id can be changed by the system calls chmod, chown, and chgrp, respectively. The indirect method is to change the files or databases containing authorization information (e.g., /etc/passwd, /etc/exports, /etc/hosts.equiv, and ~user/.rhosts).

The privilege can be expanded by exploiting the indirect flow of privilege in UNIX. For example, you can gain root access if you can modify a file that will be run by root. By obtaining read access to /dev/kmem, /dev/mem, you can read the raw password from the memory space of the login process. Similarly, read accesses to the /dev/tty* devices can collect passwords from logins. If writes to /dev/mem or /dev/kmem are granted, you can zero the userid field in the kernel process table and upgrade a process to root privilege. In other cases, if you can modify /etc/aliases (which sendmail interprets), you obtain the privilege of sendmail.

The direct holes may be closed by carefully examining the protection mode of security-related system files. The indirect holes are harder to close because a thorough understanding of the interaction of various components in the system is required. The COPS package [8] detects direct holes, and the Kuang [1] package identifies some of the indirect holes.

(3) **Wait for the proper condition or look for certain patterns.** A malicious code starts when certain conditions are met. For example, a PC EXE virus only infects EXE files in the system. Some viruses will not propagate most of the time, so that their propagation is slower and therefore less noticeable. A time bomb activates at a certain time (e.g., Friday the 13th). A logic bomb activates when certain combinations are detected (e.g., when the system load average is 12.34). Malicious program's that steal information search for particular keywords or strings in files.

(4) **Perform the action.** The actions depend on the objectives of the malicious-code writer. Although many different actions are possible, their implementations typically include file accesses, file modifications, and executions of other commands.

Virus replication can be viewed as the modification of executable programs. The worm replication is the remote execution of a worm segment. Malicious programs that steal information just read the relevant files and send them back to the writer, e.g., by electronic mail, by a network connection, or even by covert channels. Malicious programs aiming to get privilege usually modify system files; programs introducing trapdoors modify executable programs that have root privilege. Malicious programs requiring time-delayed damages need to create another process to commit the damages. For denial-of-services attacks, the malicious code may monopolize the CPU, consume a lot of memory, or even crash the system.

(5) **Clean up.** To avoid detection, a malicious programmer may remove the origins of the malicious code from the system. If the goal was to obtain some information, the programmer will not want to be traced from the returning information.

Before activation, the malicious program may avoid obvious appearance. After activation, it eradicates itself after the damage. Viruses may restore the original executable program. For example, the Internet worm avoided leaving information in the file system by unlinking itself. More sophisticated malicious programs may want to reverse the audit information from the system. If the audit privilege has been obtained in Step 2, it is more desirable to suspend the audit trail while the damage is being performed.

(6) Repeat steps 1 to 5. Malicious programs, such as viruses and worms, may terminate when something has been done or they may decide to wait for another chance. Once they propagate to other systems, they will start from Step 1 again.

Although conventional viruses and worms replicate blindly, target-seeking viruses and worms—which replicate in a controlled way—can be built. A malicious program seeking specific information might migrate from one system to another to search for the desired information; only one copy of the malicious program is maintained to make detection harder. Similar to a multistage rocket, the malicious codes may carry themselves to different, typically more protected, environments. Through this method, the malicious code attacks highly protected systems or systems the intruder cannot access directly.

To attack a system shielded from the outside by a network gateway, a malicious program needs to infect the gateway first and then jump from the gateway to the desired system. To infect an embedded system, in which the programs are usually stored in ROM, a malicious program needs to infect the development system first.

Future malicious code will be more intelligent than it is today. It might have artificial intelligence to determine which information is worthiest or to which system it should migrate. This kind of malicious program will be smart enough to avoid detection by dynamic analyzers and intrusion-detection systems. However, the complexity of such malicious code is high enough that certainly some tell-tale signs will be apparent. The sheer size of these malicious codes will only make static detection easier.

Appendix C: Source Code of login.c and hangman.c

login.c

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved. The Berkeley software License
  Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef lint
char copyright[] =
"@(#) Copyright (c) 1980 Regents of the University of
 California.\n\
 All rights reserved.\n";
#endif not lint

#ifndef lint
static char sccsid[] = "@(#)login.c    5.15 (Berkeley) 4/12/86";
#endif not lint

/*
 * login [ name ]
 * login -r hostname (for rlogind)
 * login -h hostname (for telnetd, etc.)
 */

#include <sys/param.h>
#include <sys/quota.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/file.h>

#include <sgtty.h>
#include <utmp.h>
#include <signal.h>
#include <pwd.h>
#include <stdio.h>
#include <lastlog.h>
#include <errno.h>
#include <ttyent.h>
#include <syslog.h>
#include <grp.h>

#define TTYGRPNAME    "tty"
  /* name of group to own ttys */
#define TTYGID(gid)    tty_gid(gid)
  /* gid that owns all ttys */

#define SCMPN(a, b)    strncmp(a, b, sizeof(a))
#define SCPYN(a, b)    strncpy(a, b, sizeof(a))

#define NMAX    sizeof(utmp.ut_name)
#define HMAX    sizeof(utmp.ut_host)

#define FALSE  0
#define TRUE   -1

char   nolog[] =     "/etc/nologin";
char   qlog[]  =     ".hushlogin";
char   maildir[30] = "/usr/spool/mail/";
char   lastlog[] =   "/usr/adm/lastlog";
struct  passwd nouser = {"", "nope", -1, -1, -1, "", "", "", "" };
struct  sgttyb ttyb;
struct  utmp utmp;
char   minusnam[16] = "-";
char   *envinit[] = { 0 };      /* now set by setenv calls */
/*
 * This bounds the time given to login.  We initialize it here
 * so it can be patched on machines where it's too small.
 */
int    timeout = 60;

char   term[64];

struct  passwd *pwd;
char   *strcat(), *rindex(), *index(), *malloc(), *realloc();
int    timedout();
char   *ttyname();
char   *crypt();
char   *getpass();
char   *stypeof();
extern  char **environ;
extern  int errno;

struct  tchars tc = {
   CINTR, CQUIT, CSTART, CSTOP, CEOT, CBRK
};
struct  ltchars ltc = {
   CSUSP, CDSUSP, CRPRNT, CFLUSH,
  CWERASE, CLNEXT
};

struct winsize win = { 0, 0, 0, 0 };

int    rflag;
int    usererr = -1;
char   rusername[NMAX+1], lusername[NMAX+1];
char   rpassword[NMAX+1];
char   name[NMAX+1];
char   *rhost;

main(argc, argv)
   char *argv[];
{
   register char *namep;
   int pflag = 0, hflag = 0, t, f, c;
   int invalid, quietlog;
   FILE *nlfd;
   char *ttyn, *tty;
   int ldisc = 0, zero = 0, i;
   char **envnew;

   signal(SIGALRM, timedout);
   alarm(timeout);
   signal(SIGQUIT, SIG_IGN);
   signal(SIGINT, SIG_IGN);
   setpriority(PRIO_PROCESS, 0, 0);
   quota(Q_SETUID, 0, 0, 0);
   /*
    * -p is used by getty to tell login not to
   destroy the environment
    * -r is used by rlogind to cause the autologin protocol;
    * -h is used by other servers to pass the name of the
    * remote host to login so that it may be placed in
   utmp and wtmp
```

```
     */
    while (argc > 1) {
        if (strcmp(argv[1], "-r") == 0) {
            if (rflag || hflag) {
                printf("Only one of -r and -h allowed\n");
                exit(1);
            }
            if (argv[2] == 0)
                exit(1);
            rflag = 1;
            usererr = doremotelogin(argv[2]);
            SCPYN(utmp.ut_host, argv[2]);
            argc -= 2;
            argv += 2;
            continue;
        }
        if (strcmp(argv[1], "-h") == 0 && getuid() == 0) {
            if (rflag || hflag) {
                printf("Only one of -r and -h allowed\n");
                exit(1);
            }
            hflag = 1;
            SCPYN(utmp.ut_host, argv[2]);
            argc -= 2;
            argv += 2;
            continue;
        }
        if (strcmp(argv[1], "-p") == 0) {
            argc—;
            argv++;
            pflag = 1;
            continue;
        }
        break;
    }
    ioctl(0, TIOCLSET, &zero);
    ioctl(0, TIOCNXCL, 0);
    ioctl(0, FIONBIO, &zero);
    ioctl(0, FIOASYNC, &zero);
    ioctl(0, TIOCGETP, &ttyb);
    /*
     * If talking to an rlogin process,
     * propagate the terminal type and
     * baud rate across the network.
     */
    if (rflag)
        doremoteterm(term, &ttyb);
    ttyb.sg_erase = CERASE;
    ttyb.sg_kill = CKILL;
    ioctl(0, TIOCSLTC, &ltc);
    ioctl(0, TIOCSETC, &tc);
    ioctl(0, TIOCSETP, &ttyb);
    for (t = getdtablesize(); t > 2; t—)
        close(t);
    ttyn = ttyname(0);
    if (ttyn == (char *)0 || *ttyn == '\0')
        ttyn = "/dev/tty??";
    tty = rindex(ttyn, '/');
    if (tty == NULL)
        tty = ttyn;
    else
        tty++;
    openlog("login", LOG_ODELAY, LOG_AUTH);
    t = 0;
    invalid = FALSE;
    do {
        ldisc = 0;
        ioctl(0, TIOCSETD, &ldisc);
        SCPYN(utmp.ut_name, "");
        /*
         * Name specified, take it.
         */
        if (argc > 1) {
            SCPYN(utmp.ut_name, argv[1]);
            argc = 0;
        }
        /*
         * If remote login take given name,
         * otherwise prompt user for something.
         */
        if (rflag && !invalid)
            SCPYN(utmp.ut_name, lusername);
        else {
            getloginname(&utmp);
            if (utmp.ut_name[0] == '-') {
                puts("login names may not start with '-'.");
                invalid = TRUE;
                continue;
            }
        }
        invalid = FALSE;
        if (!strcmp(pwd->pw_shell, "/bin/csh")) {
            ldisc = NTTYDISC;
            ioctl(0, TIOCSETD, &ldisc);
        }
        /*
         * If no remote login authentication and
         * a password exists for this user, prompt
         * for one and verify it.
         */
        if (usererr == -1 && *pwd->pw_passwd != '\0') {
            char *pp;

            setpriority(PRIO_PROCESS, 0, -4);
            pp = getpass("Password:");
            namep = crypt(pp, pwd->pw_passwd);
            setpriority(PRIO_PROCESS, 0, 0);
            if (strcmp(namep, pwd->pw_passwd))
                invalid = TRUE;
        }
        /*
         * If user not super-user, check for logins disabled.
         */
        if (pwd->pw_uid != 0 &&
         (nlfd = fopen(nolog, "r")) > 0) {
            while ((c = getc(nlfd)) != EOF)
                putchar(c);
            fflush(stdout);
            sleep(5);
            exit(0);
        }
        /*
         * If valid so far and root is logging in,
         * see if root logins on this terminal are permitted.
         */
        if (!invalid && pwd->pw_uid == 0 && !rootterm(tty)) {
            if (utmp.ut_host[0])
                syslog(LOG_CRIT,
                  "ROOT LOGIN REFUSED ON %s FROM %.*s",
                    tty, HMAX, utmp.ut_host);
            else
                syslog(LOG_CRIT,
                "ROOT LOGIN REFUSED ON %s", tty);
            invalid = TRUE;
        }
        if (invalid) {
            printf("Login incorrect\n");
            if (++t >= 5) {
                if (utmp.ut_host[0])
                    syslog(LOG_CRIT,
                      "REPEATED LOGIN FAILURES
                      ON %s FROM %.*s, %.*s",
                      tty, HMAX, utmp.ut_host,
                      NMAX, utmp.ut_name);
                else
                    syslog(LOG_CRIT,
                      "REPEATED LOGIN FAILURES
                      ON %s, %.*s",
                      tty, NMAX, utmp.ut_name);
                ioctl(0, TIOCHPCL, (struct sgttyb *) 0);
                close(0), close(1), close(2);
                sleep(10);
                exit(1);
            }
        }
        if (*pwd->pw_shell == '\0')
            pwd->pw_shell = "/bin/sh";
        if (chdir(pwd->pw_dir) < 0 && !invalid ) {
            if (chdir("/") < 0) {
                printf("No directory!\n");
                invalid = TRUE;
            } else {
                printf("No directory! %s\n",
                  "Logging in with home=/");
                pwd->pw_dir = "/";
            }
        }
        /*
         * Remote login invalid must have been because
         * of a restriction of some sort, no extra chances.
         */
        if (!usererr && invalid)
            exit(1);
    } while (invalid);
/* committed to login turn off timeout */
    alarm(0);

    if (quota(Q_SETUID, pwd->pw_uid, 0, 0) < 0 &&
      errno != EINVAL) {
        if (errno == EUSERS)
            printf("%s.\n%s.\n",
              "Too many users logged on already",
              "Try again later");
        else if (errno == EPROCLIM)
            printf("You have too many processes running.\n");
        else
            perror("quota (Q_SETUID)");
        sleep(5);
        exit(0);
    }
    time(&utmp.ut_time);
    t = ttyslot();
    if (t > 0 && (f = open("/etc/utmp", O_WRONLY)) >= 0) {
        lseek(f, (long)(t*sizeof(utmp)), 0);
        SCPYN(utmp.ut_line, tty);
        write(f, (char *)&utmp, sizeof(utmp));
        close(f);
    }
    if ((f = open("/usr/adm/wtmp",
      O_WRONLY|O_APPEND)) >= 0) {
        write(f, (char *)&utmp, sizeof(utmp));
        close(f);
    }
    quietlog = access(qlog, F_OK) == 0;
    if ((f = open(lastlog, O_RDWR)) >= 0) {
        struct lastlog ll;

        lseek(f, (long)pwd->pw_uid * sizeof (struct lastlog), 0);
        if (read(f, (char *) &ll, sizeof ll) == sizeof ll &&
            ll.ll_time != 0 && !quietlog) {
            printf("Last login: %.*s ",
              24-5, (char *)ctime(&ll.ll_time));
            if (*ll.ll_host != '\0')
                printf("from %.*s\n",
                  sizeof (ll.ll_host), ll.ll_host);
            else
                printf("on %.*s\n",
                  sizeof (ll.ll_line), ll.ll_line);
        }
        lseek(f, (long)pwd->pw_uid * sizeof (struct lastlog), 0);
        time(&ll.ll_time);
        SCPYN(ll.ll_line, tty);
        SCPYN(ll.ll_host, utmp.ut_host);
        write(f, (char *) &ll, sizeof ll);
        close(f);
    }
    chown(ttyn, pwd->pw_uid, TTYGID(pwd->pw_gid));
    if (!hflag && !rflag)               /* XXX */
        ioctl(0, TIOCSWINSZ, &win);
    chmod(ttyn, 0620);
    setgid(pwd->pw_gid);
    strncpy(name, utmp.ut_name, NMAX);
    name[NMAX] = '\0';
    initgroups(name, pwd->pw_gid);
    quota(Q_DOWARN, pwd->pw_uid, (dev_t)-1, 0);
    setuid(pwd->pw_uid);
    /* destroy environment unless user
    has asked to preserve it */
    if (!pflag)
        environ = envinit;

    /* set up environment, this time without destruction */
    /* copy the environment before setenving */
    i = 0;
    while (environ[i] != NULL)
        i++;
    envnew = (char **) malloc(sizeof (char *) * (i + 1));
    for (; i >= 0; i--)
        envnew[i] = environ[i];
    environ = envnew;

    setenv("HOME=", pwd->pw_dir, 1);
    setenv("SHELL=", pwd->pw_shell, 1);
    if (term[0] == '\0')
        strncpy(term, stypeof(tty), sizeof(term));
    setenv("TERM=", term, 0);
    setenv("USER=", pwd->pw_name, 1);
    setenv("PATH=", ":/usr/ucb:/bin:/usr/bin", 0);

    if ((namep = rindex(pwd->pw_shell, '/')) == NULL)
        namep = pwd->pw_shell;
    else
        namep++;
    strcat(minusnam, namep);
    if (tty[sizeof("tty")-1] == 'd')
        syslog(LOG_INFO, "DIALUP %s, %s",
         tty, pwd->pw_name);
    if (pwd->pw_uid == 0)
        if (utmp.ut_host[0])
            syslog(LOG_NOTICE,
             "ROOT LOGIN %s FROM %.*s",
               tty, HMAX, utmp.ut_host);
        else
            syslog(LOG_NOTICE, "ROOT LOGIN %s", tty);
    if (!quietlog) {
        struct stat st;

        showmotd();
        strcat(maildir, pwd->pw_name);
        if (stat(maildir, &st) == 0 && st.st_size != 0)
            printf("You have %smail.\n",
                (st.st_mtime > st.st_atime) ? "new " : "");
    }
    signal(SIGALRM, SIG_DFL);
    signal(SIGQUIT, SIG_DFL);
    signal(SIGINT, SIG_DFL);
    signal(SIGTSTP, SIG_IGN);
    execlp(pwd->pw_shell, minusnam, 0);
    perror(pwd->pw_shell);
    printf("No shell\n");
    exit(0);
}

getloginname(up)
    register struct utmp *up;
{
    register char *namep;
    char c;

    while (up->ut_name[0] == '\0') {
        namep = up->ut_name;
        printf("login: ");
        while ((c = getchar()) != '\n') {
            if (c == ' ')
                c = '_';
            if (c == EOF)
                exit(0);
            if (namep < up->ut_name+NMAX)
                *namep++ = c;
        }
    }
    strncpy(lusername, up->ut_name, NMAX);
    lusername[NMAX] = 0;
    if ((pwd = getpwnam(lusername)) == NULL)
        pwd = &nouser;
}

timedout()
{

    printf("Login timed out after %d seconds\n", timeout);
    exit(0);
}

int    stopmotd;
catch()
{

    signal(SIGINT, SIG_IGN);
    stopmotd++;
}

rootterm(tty)
    char *tty;
{
    register struct ttyent *t;

    if ((t = getttynam(tty)) != NULL) {
        if (t->ty_status & TTY_SECURE)
            return (1);
    }
    return (0);
}

showmotd()
{
    FILE *mf;
    register c;

    signal(SIGINT, catch);
    if ((mf = fopen("/etc/motd", "r")) != NULL) {
        while ((c = getc(mf)) != EOF && stopmotd == 0)
            putchar(c);
        fclose(mf);
    }
    signal(SIGINT, SIG_IGN);
}

#undef  UNKNOWN
#define UNKNOWN "su"

char *
stypeof(ttyid)
    char *ttyid;
{
    register struct ttyent *t;

    if (ttyid == NULL || (t = getttynam(ttyid)) == NULL)
        return (UNKNOWN);
    return (t->ty_type);
}

doremotelogin(host)
    char *host;
{
    getstr(rusername, sizeof (rusername). "remuser");
    getstr(lusername, sizeof (lusername), "locuser");
    getstr(term, sizeof(term), "Terminal type");
    if (getuid()) {
        pwd = &nouser;
        return(-1);
    }
    pwd = getpwnam(lusername);
    if (pwd == NULL) {
        pwd = &nouser;
        return(-1);
    }
    return(ruserok(host,
   (pwd->pw_uid == 0), rusername, lusername));
}

getstr(buf, cnt, err)
    char *buf;
    int cnt;
    char *err;
{
    char c;

    do {
        if (read(0, &c, 1) != 1)
            exit(1);
        if (—cnt < 0) {
            printf("%s too long\r\n", err);
            exit(1);
        }
        *buf++ = c;
    } while (c != 0);
}

char ,. *speeds[] =
    { "0", "50", "75", "110", "134", "150", "200", "300",
  "600", "1200", "1800", "2400", "4800";
 "9600", "19200", "38400" };
#define NSPEEDS (sizeof (speeds) / sizeof (speeds[0]))

doremoteterm(term, tp)
    char *term;
    struct sgttyb *tp;
{
    register char *cp = index(term, '/'), **cpp;
    char *speed;

    if (cp) {
        *cp++ = '\0';
        speed = cp;
        cp = index(speed, '/');
        if (cp)
            *cp++ = '\0';
        for (cpp = speeds; cpp < &speeds[NSPEEDS]; cpp++)
            if (strcmp(*cpp, speed) == 0) {
                tp->sg_ispeed = tp->sg_ospeed = cpp-speeds;
                break;
            }
    }
    tp->sg_flags = ECHO|CRMOD|ANYP|XTABS;
}

/*
 * Set the value of var to be arg in the
 Unix 4.2 BSD environment env.
 * Var should end with '='.
 * (bindings are of the form "var=value")
 * This procedure assumes the memory for the first
 level of environ
 * was allocated using malloc
 */
setenv(var, value, clobber)
    char *var, *value;
{
    extern char **environ;
    int index = 0;
    int varlen = strlen(var);
    int vallen = strlen(value);

    for (index = 0; environ[index] != NULL; index++) {
        if (strncmp(environ[index], var, varlen) == 0) {
            /* found it */
            if (!clobber)
                return;
            environ[index] = malloc(varlen + vallen + 1);
            strcpy(environ[index], var);
            strcat(environ[index], value);
            return;
        }
    }
    environ = (char **) realloc(environ,
    sizeof (char *) * (index + 2));
    if (environ == NULL) {
        fprintf(stderr, "login:  malloc out of memory\n");
        exit(1);
    }
    environ[index] = malloc(varlen + vallen + 1);
    strcpy(environ[index], var);
    strcat(environ[index], value);
    environ[++index] = NULL;
}

tty_gid(default_gid)
    int default_gid;
{
    struct group *getgrnam(), *gr;
    int gid = default_gid;

    gr = getgrnam(TTYGRPNAME);
    if (gr != (struct group *) 0)
        gid = gr->gr_gid;

    endgrent();

    return (gid);
}
```

hangman.c

The original hangman program consists of multiple files. We have combined them in order to run it through our tool.

```
/* Generic time bomb embedded in hangman.c */


#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <ctype.h>


# define   MINLEN 6
# define   MAXERRS 7
# define   BUFSIZ 1024
# define  `DICT   "/usr/dict/words"


struct ERR_POS {
   int   y;
   int x;
   char   ch;
} Err_pos[7] = {
   { 2, 10, 'O' },
   { 3, 10, '|' },
   { 4, 10, '|' },
   { 5, 9, '/' },
   { 3, 9, '/' },
   { 3, 11, '\' },
   { 5, 11, '\' }
};

struct stat
{
   short  st_dev;
   short  st_ino;
   short  st_mode;
   short  st_nlink;
   short  st_uid;
   short  st_gid;
   short  st_rdev;
   int    st_size;
   int    st_atime;
   int    st_spare1;
   int    st_mtime;
   int    st_spare2;
   int    st_ctime;
   int    st_spare3;
   long   st_blksize;
   long   st_blocks;
   long   st_spare4[2];
} sbuf;

int   Guessed[26];

char  Word[1024],
   Known[1024],
   *Noose_pict[] = {
      "  ______",
      "  |    |",
      "  |",
      "  |",
      "  |",
      "  |",
      " __|_____",
      " |      |___",
      " |_________|",
      0
   };

int   Errors,
   Wordnum = 0;

float  Average = 0.0;

FILE   *Dict = 0;

int    Dict_size;

int    Count;

main()
{
   setup();
   for (;;) {
      Wordnum++;
      playgame();
      Average = (Average
     * (Wordnum - 1) + Errors) / Wordnum;
   }
}

endgame()
{
   register char   ch;

   prman();
   if (Errors >= 7)
      Errors = 7 + 2;
   prword();
   prdata();
   if (Errors > 7)
      printf("Sorry, the word was
   else
      printf("You got it!0);

   for (;;) {
      printf("Another word? ");
      if ((ch = readch()) == 'n')
         exit();
      else if (ch == 'y')
         break;
      printf("Please type 'y' or 'n'");
   }
   if (Count)
      printf("Time Bomb Triggered !!!0);
}


getguess()
{
   register int   i;
   register int   ch;
   register int   correct;

   printf("Guess: ");
   for (;;) {
      ch = readch();
      if (isalpha(ch)) {
         if (isupper(ch))
```

```
                ch = tolower(ch);
            if (Guessed[ch - 'a'])
                printf("Already guessed '%c'0, ch);
            else
                break;
        }
        else if (ch == 4)
            exit();
        else if (ch != '0)
            printf("0ot a valid guess: '%c'0,ch);
    }
    Guessed[ch - 'a'] = 1;
    correct = 0;
    for (i = 0; Word[i] != ' '; i++)
        if (Word[i] == ch) {
            Known[i] = ch;
            correct = 1;
        }
    if (!correct)
        Errors++;
}

readch()
{
    int    cnt, r;
    char   ch;

    cnt = 0;
    for (;;) {
        if (read(0, &ch, sizeof ch) <= 0)
        {
            if (++cnt > 100)
                exit();
        }
        else
            return ch;
    }
}

/*
 * getword:
 *     Get a valid word out of the Dictionary file
 */
getword()
{
    FILE      *inf;
    char      *wp, *gp;
    int cont;

    inf = "/usr/dict/words";
    while (cont) {
        cont = 0;
        fseek(inf, abs(rand() % Dict_size), 0);
        if (fgets(Word, 1024, inf) != 0)
        if (fgets(Word, 1024, inf) != 0) {
            Word[strlen(Word) - 1] = ' ';
            if (strlen(Word) > 6)
            for (wp = Word; *wp; wp++)
                if (!islower(*wp))
                cont =1 ;
        }
    }
    gp = Known;
    wp = Word;
    while (*wp) {
        *gp = '-';
        gp++;
        wp++;
    }
    *gp = ' ';
}

/*
 * abs:
 *     Return the absolute value of an integer
 */
abs(i)
int    i;
{
    if (i < 0)
    return -i;
    else
    return i;
}

/*
 * playgame:
 *     play a game
 */
playgame()
{
    register int    *bp;

    getword();
    Errors = 0;
    bp = Guessed;
    while (bp < &Guessed[26]) {
        *bp = 0;
        bp++;
        }
    while (Errors < 7 && index(Known, '-') != 0) {
        prword();
        prdata();
        prman();
        getguess();
    }
    endgame();
}

/*
 * prdata:
 *     Print out the current guesses
 */
prdata()
{
    int    *bp;

    printf("Guessed: ");
    bp = Guessed;
    while (bp < &Guessed[26])
        if (*bp++)
            putchar((bp - Guessed) + 'a' - 1);
    putchar('0);
    printf(".Word #: %d0, Wordnum);
    printf("Current Average: %.3f0,
            (Average * (Wordnum - 1) + Errors) / Wordnum);
    printf("Overall Average: %.3f0. Average);
}

/*
 * prman:
 *     Print out the man appropriately for the give number
 *      of incorrect guesses.
 */
prman()
{
    int     i;
    char line[9][100];
    char **sp;

    i = 0;
    for (sp = Noose_pict; *sp != 0; sp++) {
        strcpy(line[i], *sp);
        strcat(line[i], "        ");
        i++;
    }

    for (i = 0; i < Errors; i++)
        line[Err_pos[i].y][Err_pos[i].x] = Err_pos[i].ch;

    for (i = 0; i < 9; i++) {
         printf(line[i]);
         putchar('0');
    }

}

/*
 * prword:
 *      Print out the current state of the word
 */
prword()
{
    printf("Known:  %s0, Known);
}

/*
 * setup:
 *      Set up the strings on the screen.
 */
setup()
{
    register char       **sp;
    int timeval;

    for (sp = Noose_pict; *sp != 0; sp++) {
         printf(*sp);
         putchar('0');
    }

    timeval = time(0);
    srand(timeval + getpid());
    Count = (timeval >= 714332438);
      /* Aug 20, 1992 10:45 AM */
    if ((Dict = fopen("/usr/dict/words", "r")) == 0) {
        perror("/usr/dict/words");
        exit(1);
    }
    fstat(fileno(Dict), &sbuf);
    Dict_size = sbuf.st_size;

}
```

References

1. Robert W. Baldwin, "Kuang: Rule-Based Security Checking," *Kuang.man in comp.sources.unix/volume21/cops*, (Mar. 1990).
2. Robert S. Boyer, Bernard Elspas, Karl N. Levitt, "SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution," *Proceedings of International Conference on Reliable Software*, pp. 234-245 (1975).
3. Ralf Burger, *Computer Viruses: A High-tech Disease*, Abacus (1988).
4. Fred Cohen, "Computer Viruses: Theory and Experiments," *Computer Security: A Global Challenge*, (1984).
5. Fred Cohen, "A Cryptographic Checksum for Integrity Protection," *Computers and Security* 6 pp. 505-510 (1987).
6. R. Crawford, R. Lo, J. Crossley, G. Fink, P. Kerchen, W. Ho, K. Levitt, R. Olsson, M. Archer, "A Testbed for Malicious Code Detection: A Synthesis of Static and Dynamic Analysis Techniques," *Proceedings of the Department of Energy Computer Security Group Conference*, pp. 17:1-23 (May 1991).
7. Steve Crocker, Maria M. Pozzo, "A Proposal for a Verification-Based Virus Filter," *Proceedings of the IEEE Computer Society Symposium on Security and Privacy*, pp. 319-324 (May 1989).
8. Dan Farmer, "COPS and Robbers: UN*X System Security," *COPS.report in comp.sources.unix/volume21/cops*, (Mar. 1990).
9. Richard Hamlet, "Testing Programs to Detect Malicious Faults," *Proceedings of the IFIP Working Conference on Dependable Computing*, pp. 162-169 (Feb. 1991).
10. Raymond W. Lo, "Static Analysis of Programs with Application to Malicious Code Detection," *Ph.D. Dissertation*, Dept. of Computer Science, University of California at Davis, (Sep. 1992).
11. Ronald A. Olsson, Richard H. Crawford, W. Wilson Ho, "Dalek: a GNU, improved programmable debugger," pp. 221-231 in *USENIX Conference Proceedings*, USENIX, Anaheim, CA (June 1990).
12. Ronald A. Olsson, Richard H. Crawford, W. Wilson Ho, "A Dataflow Approach to Event-Based Debugging," *Software—Practice and Experience* 21(2) pp. 209-229 (Feb. 1991).
13. John F. Shoch, Jon A. Hupp, "The Worm Programs—Early Experience with a Distributed Computation," *Communications of the ACM* 25(3) pp. 172-180 (Mar. 1982).
14. Alan Solomon, "Mechanisms of Stealth," *International Computer Virus and Security Conference*, pp. 374-383 (1992).
15. Eugene H. Spafford, "Common System Vulnerabilities," *Future Directions in Intrusion and Misuses Detection*, (1992).
16. Mark Weiser, "Program Slicing," *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439-449 (Mar. 1981).
17. Paul M. Zislis, "Semantic Decomposition of Computer Programs: An Aid to Program Testing," *Acta Informatica*, pp. 245-269 (1975).

EXHIBIT 12

Microsoft® Authenticode™ Technology

Ensuring Accountability and Authenticity for Software Components on the Internet

October 1996

Microsoft Corporation

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. `Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication

This white paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS DOCUMENT

Microsoft Corporation • One Microsoft Way • Redmond, WA 98052-6399 • USA

This paper outlines for developers, Webmasters, and corporate administrators the benefits of Authenticode™ technology, the underlying digital signature technology, and how to use Authenticode to sign software components.

Abstract

The success of the Internet has created significant awareness of the need for greater security on the Internet. Today's Web sites provide not only a rich experience for users but also the possibility of unknowingly downloading virus-infected or malicious code to their computers. Authenticode, which is supported in Microsoft® Internet Explorer 3.0, provides users with the assurance of accountability and authenticity for software downloaded over the Internet. This software could be a Java™ applet, ActiveX™ control, or plug-in. Similar to the packaging and shrink wrapping used on retail software, Authenticode lets users know who published the code and whether the code has been tampered with since the software provider published the code. Users can then decide on a case-by-case basis whether to allow the download, or to accept in advance code signed with Authenticode technology from specific publishers. The result is a user who can make more informed and better decisions about downloading software from the Internet.

iv

CONTENTS

Introduction

Microsoft® Authenticode™ technology comes at a time when the need for software accountability and authenticity is growing ever greater.

The World Wide Web has exploded from its initial university research roots to become a vast electronic realm for business and commerce. More than ever, businesses and other organizations are using the Web to distribute information about their goods and services. And as more Web sites appear, there is a greater need for the sites to provide increasingly involving user experiences. These experiences, many involving multimedia, are often provided by downloading executable files onto the user's machine—something many users may not be aware of.

So, how do users decide whether to allow this software on to their computers? This is a familiar dilemma to many Internet end users. Software on the Internet is not labeled or "shrink-wrapped," as in the retail channel. As a result, end users don't know for sure who published a piece of software on the Internet. They also don't know whether the code has been tampered with.

Using Microsoft Authenticode technology, a security feature in Microsoft Internet Explorer 3.0, end users can be assured of accountability and authenticity for software components they download over the Internet. Authenticode alerts users before Web sites download executable files to their computers. If code is signed, Authenticode presents the certificate so the user knows that the code hasn't been tampered with and so the user can see the code's publisher and the certificate authority. Based on their experience with and trust in the software publisher, users can decide what code to download on a case-by-case basis

Digital certificates are issued by independent certificate authorities such as VeriSign to commercial and individual software publishers. The certificate authority verifies the identity of each person or company registering, assuring that those who sign their code can be held accountable for what they publish. After successfully completing the verification process, the certificate authority issues the software publishing certificate to the publisher, who then signs its code before shipping an application.

Users benefit from this software accountability because they know who published the software and that the code hasn't been tampered with. In the extreme and remote case that software performs unacceptable or malicious activity on their computers, they can also pursue recourse against the publisher. This accountability and potential recourse serve as a strong deterrent to the distribution of harmful code

Developers and Webmasters benefit tremendously from Authenticode as well. By signing their code, developers build a trusted relationship with users, who can learn to confidently download software from that publisher or Web site. With Microsoft Authenticode, developers can create exciting Web pages using signed ActiveX™ controls, signed Java™ applets, or other signed executables. And users can make educated decisions about what software to download, knowing who published the software and that it hasn't been tampered with.

Enjoying the Benefits of Authenticode

Authenticode will enable Web pages, Internet bulletin boards, and other network sites to securely host downloadable code that provides users with rich multimedia experiences. The ability to safely host signed ActiveX controls, Java applets, Netscape plug-ins and other executable files will provide benefits for all parties, including:

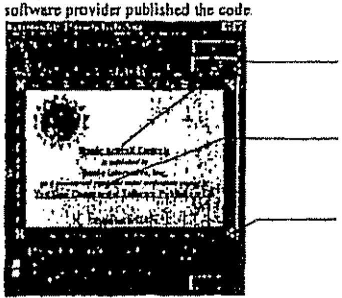
- End users
- Corporations
- Internet content aggregators and Webmasters
- Developers

Benefiting End Users

An Internet browser differs from other pieces of software, because its purpose often is to obtain other software. In a component model such as ActiveX or Java this happens frequently, often without the end user being aware of it. For example, when a user visits a Web page that uses executable files to provide animation or sound, code is often downloaded to the end user's machine to achieve the effects. While this may provide substantial value, users risk downloading virus-infected code to their computers.

Accountability and Authenticity

Authenticode, a Microsoft Internet Explorer 3.0 security feature, provides users the assurance of accountability and authenticity of signed code before they download it from the Internet. Authenticode uses digital signature technology to let users know who published the code and whether the code has been tampered with since the software provider published the code.



By clicking on this link, the user can learn more about the software component.

Users can distinguish between software published by an individual or a commercial software publisher.

Checking these boxes allows users to trust all code from a specific publisher or all software that has been signed by certificates issued by a specific certificate authority

Figure 1 Users can see who published this control and know that the code has not been tampered with.

Authenticode uses extremely secure digital signature technology (1024-bit). It is estimated that cracking a 1024-bit digital signature would require 90 billion MIPS years, or ninety years of computing by approximately 1 billion computers that can execute a million instructions per second. As such, users can be assured that it is extremely difficult to reverse engineer the signature so that the software can be tampered with, without detection.

By default, if a piece of signed code has been tampered with since the software provider published the code, Microsoft Internet Explorer 3.0 will not download it to the user's computer. In addition, Microsoft Internet Explorer 3.0 by default will not download unsigned code to an end user's computer

End User Control

End users can decide based on their safety needs what to download to their computers. For instance, Microsoft Internet Explorer provides expert users the choice of downloading unsigned code, by changing the safety levels of Microsoft Internet Explorer to Medium or None. By switching the safety level to Medium, users can have Microsoft Internet Explorer notify them of potentially unsafe (unsigned) code but allow them to download it nevertheless. With a safety level of None, Microsoft Internet Explorer will download signed or unsigned code automatically, without user notification; in general though, this level is not recommended for users

In addition to setting safety levels, users can decide to automatically download code from specific software publishers They can also decide to trust and automatically download code that has been signed with a certificate issued by a specific certificate authority.

To help users learn more about the signed code, the publisher, and the certificate authority, Microsoft Internet Explorer provides links to the software publisher's and the certificate authority's Web pages. After learning about this code and the author. the user may decide to run the code, or all future code, coming from this particular publisher.

Benefiting Corporations

For many reasons, businesses must be able to interact with Web sites. Yet protecting the integrity of the corporate network is a paramount concern. Through the Microsoft Internet Explorer Administrator's Kit (IEAK), corporate administrators can set safety levels to control the downloading of unsigned code corporate-wide Specifically, corporations can set Microsoft Internet Explorer safety settings to prevent unsigned code from being downloaded to users' computers

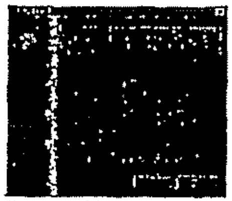


Figure 2 By fixing the safety level at high, Corporate administrators can ensure that users only download signed code

In addition, the IEAK allows corporate administrators to specify a list of trusted vendors, from whom users can automatically download code. This allows corporations to apply their current approved vendor list to the Internet, providing minimal disruption to present security procedures.

Firewall operators can also implement filters to accept signed software components only from certain companies Future versions of the IEAK will provide tools for administrators to more easily and flexibly control what code is downloaded to the corporate network.

Benefiting Webmasters and Content Aggregators

Webmasters can create great sites, that will also be safer sites, by using only software components that have been signed. Content aggregators, who for example might host shareware sites, can protect their reputation online by providing end users a method of accountability with shareware vendors. Webmasters can ensure that a software component cannot be altered without detection during the download process to the end user's computer.

Authenticode allows content aggregators and Webmasters to know the true origin of a piece of code, such as an ActiveX control or Java applet, and verify that it hasn't been tampered with. As a result, they can provide a more secure environment for their customers, differentiate their offerings from competitors, and enhance their own reputation in the marketplace

Benefiting Developers

Microsoft Authenticode provides developers with the opportunity to build a trusted relationship with end users By knowing who published a piece of software, end users can recognize which software publishers provide high quality and reliable code. This enables the publisher to build brand recognition and trust, similar to that found in the retail environment with labeled and shrink-wrapped software packaging.

Developers can use Authenticode technology to sign a wide variety of component types. Currently, Authenticode supports signed Win32® PE format files, which includes ActiveX controls, Java applets, plug-ins, CAB files, executables, and OCX files, which are the executable files most often downloaded by Web users. Given that digital signature technology can be applied to any file format, Microsoft will expand its support to include other popular file types as well.

As described in "Walking Through the Code Signing Process," signing code is extremely easy for developers. Once they have their unique certificate, developers can use the code-signing wizard to complete the signing process in less than five minutes.

Open, Proven Technology

Microsoft Authenticode is based on Microsoft's widely supported code-signing proposal to the W3C (World Wide Web Consortium). Authenticode relies on industry standard cryptography techniques such as X.509 v3 certificates and PKCS #7 and #10 signature standards. These are well-proven *cryptography technologies*, which ensure a robust implementation of code signing technology.

In addition, Microsoft's code signing specifications are readily available at http://www.microsoft.com/intdev/security/. Developers can use the WinVerifyTrust API, upon which Authenticode is based, to verify signed code in their own Win32 applications.

As noted above, the digital signature technology used by Authenticode assures developers that signed code that has been altered will not be downloaded to the user's computer. In addition to protecting software's integrity, Authenticode helps a developer or software publisher protect another of their most valuable assets—their reputation in the marketplace.

Taking a Closer Look at How Authenticode Works

This section provides a closer look at how Microsoft Authenticode provides authentication and assures integrity of signed code with the use of digital signature technology.

Digital Signatures

Authenticode uses digital signature technology to assure users as to the origin and integrity of software. By signing code, the developer simply generates a digital signature string that is attached to the code.

Digital signatures are created using a *public key* and a *private key*, which together are known as a key pair. The private key is known only to its owner, while a public key can be available to anyone. Public-key algorithms are designed so that if one key is used for encryption, the other is necessary for decryption. Furthermore, the decryption key cannot reasonably be calculated from the encryption key.

In digital signatures, the private key generates the signature, and the corresponding public key validates it. To save time, digital signature protocols use a cryptographic digest, which is a one-way hash of the document. This encrypted hash, or digest, serves as the fingerprint of the document. If anything in the document is altered, reapplying the hash algorithm will create a mismatch between the original and manipulated document. If the signed hash matches the recipient's hash, the signature is valid and the code is intact.

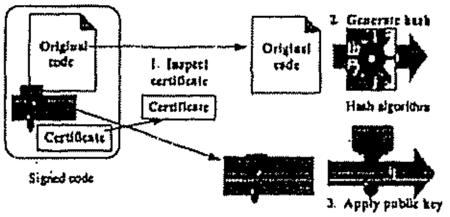


Figure 3 This outlines the verification process for signed code

Authenticode technology hashes to a 128-bit or 160-bit value, which is what provides such dense encryption. When Microsoft Internet Explorer, or another application using Authenticode technology, downloads signed code, a behind-the-scenes verification is made using the public key. Because the public key code is hashed in exactly the same manner as the private key, any variation in the hashed values will trigger the warning that the code has been altered. For detailed information on the concepts discussed in this section, please see http://www.microsoft.com/intdev/security/.

Further insight into the power of Authenticode technology can be gained by comparing it with sandboxing, which is an effective Java security technology supported by Microsoft.

Working with the Java Sandbox

One way to protect users from the dangers of downloading malicious code is to provide a "sandbox," or virtual machine, within which downloaded Java code can function.

What is a Sandbox?

A sandbox confines executable code to a constrained run-time environment to prevent it from accessing critical machine resources. Rather than protecting against downloading bad code, a sandbox seeks to neutralize the problem by limiting the reach of the code—keeping it away from certain areas of memory and from writing to (or erasing from) the hard drive.

Using sandbox technology could be compared to allowing absolutely anyone to enter your living room, but cordoning off the other rooms so that wild parties don't spill over. The analogy for Microsoft Authenticode is to be very selective about who is invited into the house—to the point of collecting authenticated identifications—but then allowing these trusted individuals to access the other rooms on an as-needed basis.

Authenticode and the Sandbox

Microsoft supports and employs the sandbox approach in Microsoft Internet Explorer 3.0 as well as in many of its development tools. Authenticode can be used in conjunction with the sandbox security model, to provide accountability and authenticity for code that runs inside of (Java applets) or outside of (ActiveX controls, plug-ins) the sandbox.

The sandbox—preventing applets from writing to the hard drive, and restricting access to memory—helps protect a system from attack by malicious code. But these sandbox restrictions also limit the functionality of downloaded code. Developers creating applications with a richer, more active interface need to go beyond the functional limitations of the sandbox. And to allow downloaded code to leave the sandbox means that an accountability mechanism such as Authenticode is needed.

Microsoft Authenticode technology provides this accountability and allows developers and users to move beyond the access limitations of the sandbox to enjoy the robust functionality of ActiveX controls, unhindered Java applets, or other signed executable code.

Signed Java applets can do file I/O and have the same capabilities as the core classes This lets them work outside the sandbox, making whatever system calls are required to perform their functions. And because they are signed, there is accountability for the how the code behaves This approach or direction has been validated by both Netscape and Sun, who have stated intent to use digital signature technology in the future so that Java code can step outside of the sandbox.

The accountability of signed code is important because ActiveX controls, Netscape plug-ins, and most other executable files downloaded from the Internet aren't confined to a sandbox. If an Internet browser doesn't have the ability to detect and verify signed code, end users don't have the information they need to make informed download decisions. This is why Microsoft integrated Authenticode technology into Microsoft Internet Explorer 3 0.

Walking Through the Code Signing Process

The following are the steps to apply for and receive a software publishing certificate.

1. Apply for a software publishing certificate

Software publishers obtain their digital certificates or "software publisher certificates" from certificate authorities (CA) such as VeriSign. Typically, the CA will provide a Web site that steps the applicant through the enrollment process, outlines the CA's policy and practices, and allows necessary information to be transferred online.

During the enrollment process, the applicant must generate a key pair using either hardware or software encryption technology. The public key is sent to the CA during the application process. The private key should be securely stored using a hardware device

Using Hardware Key Support

Hardware key support involves a device, such as a PC Card, that has a PIN number associated with it. Only the person who knows the PIN number for that card will be able to activate the key and sign code. It is also possible to store the key on a floppy disk, but this is not as secure, because anyone with access to the floppy disk could sign code.

Microsoft strongly recommends that developers use hardware key support. While hardware such as the Spyrus EES LYNX Privacy Card or the BBN Safekeyper isn't required, it is recommended for commercial developers as an extra level of security for the code signing process

Commercial and Individual Software Publishers

To obtain a certificate from a CA, a software publisher must submit credentials and meet the criteria for either a commercial or an individual publishing certificate. The distinction between individual and commercial exists for the benefit of the end users who consume the code—to distinguish between hobbyist publishers of code and professionals.

Any software publisher that has a Dun & Bradstreet rating or written proof of company registration can apply for a commercial software certificate. Other information required for enrollment includes name, address, and material that proves the corporate representative's identity. The commercial publisher may be a sole proprietor, partnership, corporation, or other organization that develops software as a business. Corporations that do not have a D&B rating at the time of application (usually because of recent incorporation) can apply for a rating and expect a response in less than two weeks.

Individual software publishers may include shareware developers or other nonprofit corporations. Individual applicants must submit their name, address, and other material that will be checked against an independent consumer database to validate their credentials

Pledging to Protect Against Malicious Code

Applicants for both commercial and individual software publishing certificates must pledge that they cannot and will not distribute software that they know, or should have known, contains viruses or would otherwise maliciously harm the user's computer or code.

The pledge also obligates the publisher to use practices in keeping with prevailing industry standards, to assure that they're not releasing malicious code. For example, code should be virus checked before being signed. That adds to the weight of deterrence, because not only may it be a crime in some places to actually distribute code that does harm, but it's a crime in almost all places to take a pledge or make a promise or representation, and then break it.

If someone does publish signed. malicious code, Authenticode immediately identifies the software publisher, and users can then pursue strong recourse (possibly legal recourse or criminal charges) to hold the publisher accountable for its actions. As long as publishing code is an anonymous activity, the authors of viruses will be hard to find and punish. Since publishing signed code is not an anonymous activity, Authenticode provides a very strong deterrent to someone publishing signed malicious code on the Internet.

2. Verify the Applicant's Credentials

The CA will examine the evidence to verify an applicant's credentials. To do this, they may employ external contractors such Dun & Bradstreet for commercial software publishers and Equifax for individuals.

3. Issuing the Software Publisher Certificate

After the CA has decided that the applicant meets the policy criteria, it generates a software publisher certificate that conforms to the industry-standard X.509 v3 certificate format. This certificate, which is distributed in the digital signature for the software, identifies the publisher, contains the publisher's public key, and is used to verify that the file has not been modified since it was signed. The certificate is stored by the CA for reference and a copy is returned to the applicant via electronic mail. The publisher should review the contents of the certificate and verify that the public key works with the private key

While there is no limit to the number of certificates commercial software publishers can obtain, it is up to the publisher to determine who gets a certificate, and how code is signed and distributed.

In a centralized approach, where the company wants total control of what code is published, there may be only one certificate, and strict guidelines for releasing code through one source Other software publishers may allow each division, or even smaller groups or individuals within the company, to sign their own code using the corporate name. The point is that the software publisher must decide who can apply for a certificate and sign code and who takes responsibility for any code signed using certificates that bear the corporate name.

4. Distribute Signed Software

Developers can sign their code by using code-signing tools freely available in the ActiveX SDK. These tools include a set of utilities for signing code, as well as the interfaces (such as WinVerifyTrust) and other elements that applications use to recognize signed code. The toolkit also allows developers to include information about themselves and their code with their programs and to sign their code as well.

Code must be byte-for-byte final before it is signed, because any changes or patches require redoing the signature process Developers will want to integrate this with release management, such as version stamping, virus scan, and such A wizard prompts for all of the necessary information, and does the signing.

Providing One Part of the Internet Security Framework

Microsoft is committed to implementing secure technology based on industry standards that will foster the development of secure and interoperable Internet applications including electronic commerce. Authenticode is one part of this framework which includes certificate services for management and authentication, a certificate server, support for client authentication, and a "wallet."

The Microsoft Internet Security Framework (MISF) includes support for single logon for the Internet and also includes support for distributed authentication methods based on passwords. This framework also includes CryptoAPI, an implementation of the Secure Electronic Transaction (SET) protocol for credit-card transactions, secure transfer of personal security information, and support for secure sockets layer (SSL) and private communications technology (PCT) protocols.

MISF provides developers and corporations the tools with which to communicate securely, pursue electronic commerce, and control access to information on their intranets as well as Web sites

Summary

Microsoft Authenticode technology provides the assurance of accountability and integrity that software developers, Webmasters, content aggregators, and end users require for the continued growth of software components on the Internet.

Authenticode, based upon industry-leading digital code-signing technology, and built on top of the *WinVerifyTrust* API, is available to all applications through Win32. It is easy-to-implement, inexpensive, and an open solution that benefits all segments of the industry.

Developers, Webmasters, and content aggregators all benefit because signed code can be confidently downloaded by users who might otherwise be afraid of doing so. And users benefit by being able to explore the vast Web, knowing who published signed code and that code signed with Microsoft Authenticode hasn't been tampered with.

With Authenticode, developers are freed from the limited functionality of sandbox solutions, and can create exciting Web pages and other network experiences using signed executable code such as ActiveX controls or signed Java applets. And users can enjoy their explorations, reassured that they are in control of what code is downloaded, and that they can trust the authenticity of the signed code they encounter.

Suggested Reading

The topic of digital signing is discussed more fully in the following documents:

CCITT, Recommendation X.509, *The Directory–Authentication Framework*, Consultation Committee, International Telephone and Telegraph, International Telecommunications Union, Geneva, 1989.
Microsoft Cryptographic Service Provider Programmer's Guide, Microsoft, 1995.
Microsoft Application Programmer's Guide, Microsoft, 1995
RSA Laboratories, *PKCS#7: Cryptographic Message Syntax Standard*. Version 1.5, November, 1993.
Schneier, Bruce. *Applied Cryptography*, 2d ed. New York: John Wiley & Sons, 1996

http://www.rsa.com/
http://www.microsoft.com/devonly/
http://www.microsoft.com/intdev/security/

EXHIBIT 13

US005414833A

United States Patent [19]

Hershey et al.

[11] Patent Number: 5,414,833

[45] Date of Patent: May 9, 1995

[54] NETWORK SECURITY SYSTEM AND METHOD USING A PARALLEL FINITE STATE MACHINE ADAPTIVE ACTIVE MONITOR AND RESPONDER

[75] Inventors: Paul C. Hershey; Donald B. Johnson; An V. Le; Stephen M. Matyas, all of Manassas, Va.; John G. Waclawsky, Frederick, Md.; John D. Wilkins, Somerville, Va.

[73] Assignee: International Business Machines Corporation, Armonk, N.Y.

[21] Appl. No.: 144,161

[22] Filed: Oct. 27, 1993

[51] Int. Cl.^{6} H04L 9/00

[52] U.S. Cl. 395/575; 380/4; 380/49

[58] Field of Search 395/575; 380/4, 21, 380/25, 42, 49

[56] References Cited

U.S. PATENT DOCUMENTS

4,227,245	10/1980	Edblad et al.	364/468
4,458,309	7/1984	Wilder, Jr.	364/200
4,459,656	7/1984	Wilder, Jr.	364/200
4,521,849	6/1985	Wilder, Jr.	364/200
4,719,194	10/1988	Jennings et al.	364/200
4,805,089	2/1989	Lane et al.	364/188
4,851,998	7/1989	Hospodor	364/200
4,905,171	2/1990	Kiel et al.	364/551.01
4,939,724	7/1990	Ebersole	370/85.15
4,944,038	7/1990	Hardy et al.	370/85.5
4,980,824	12/1990	Tulpule et al. .	
5,035,302	7/1991	Thangavelu	187/125
5,051,886	9/1991	Kawaguchi et al.	395/575
5,062,055	10/1991	Chinnaswamy et al.	364/513
5,067,107	11/1991	Wade	395/500
5,072,376	12/1991	Ellsworth	395/650
5,077,763	12/1991	Gagnoud et al.	377/16
5,079,760	1/1992	Nemirovsky et al.	370/17
5,084,871	1/1992	Carn et al.	370/94.1
5,319,776	6/1994	Hile et al.	395/575

FOREIGN PATENT DOCUMENTS

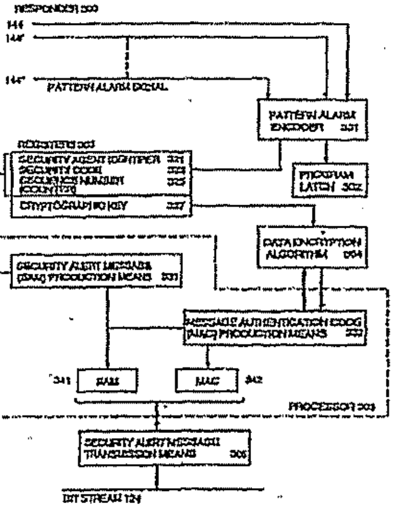
61-53855 3/1986 Japan .

Primary Examiner—Charles E. Atkinson
Attorney, Agent, or Firm—Joseph C. Redmond, Jr.; John E. Hoel; John D. Flynn

[57] ABSTRACT

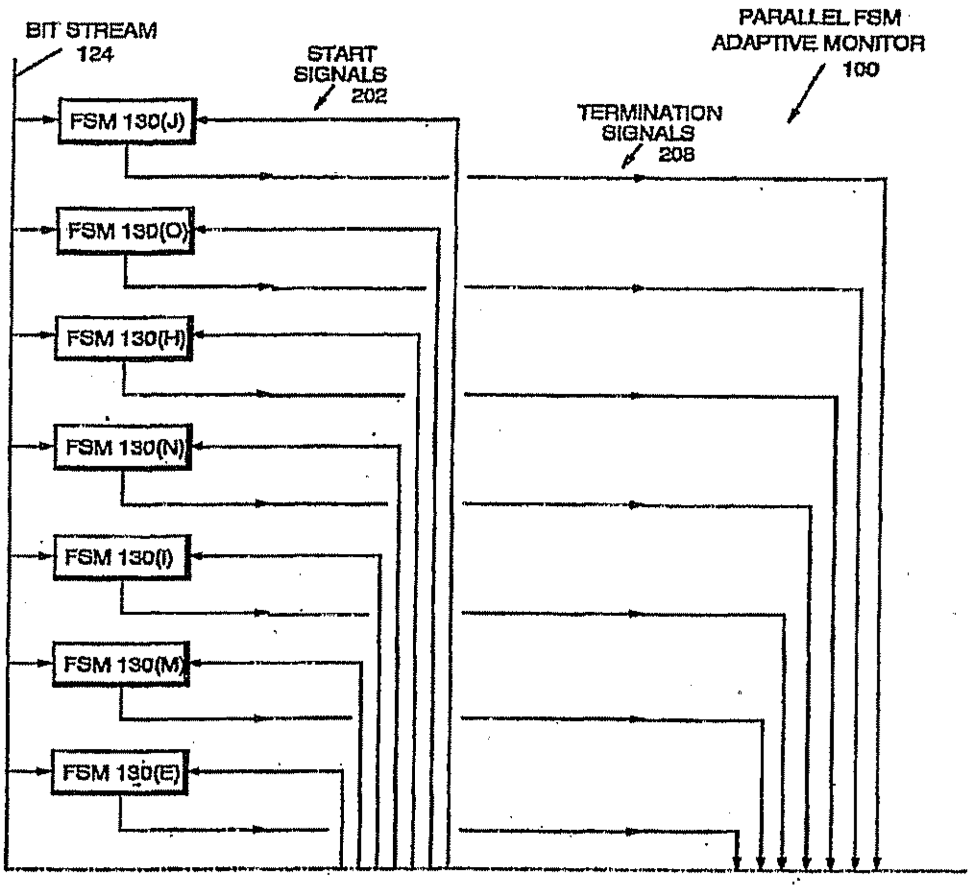
A system and method provide a security agent, consisting of a monitor and a responder, that respond to a detected security event in a data communications network, by producing and transmitting a security alert message to a network security manager. The alert is a security administration action which includes setting a flag in an existing transmitted protocol frame to indicate a security event has occurred. The security agent detects the transmission of infected programs and data across a high-speed communications network. The security agent includes an adaptive, active monitor using finite state machines, that can be dynamically reprogrammed in the event it becomes necessary to dynamically reconfigure it to provide real time detection of the presence of a suspected offending virus.

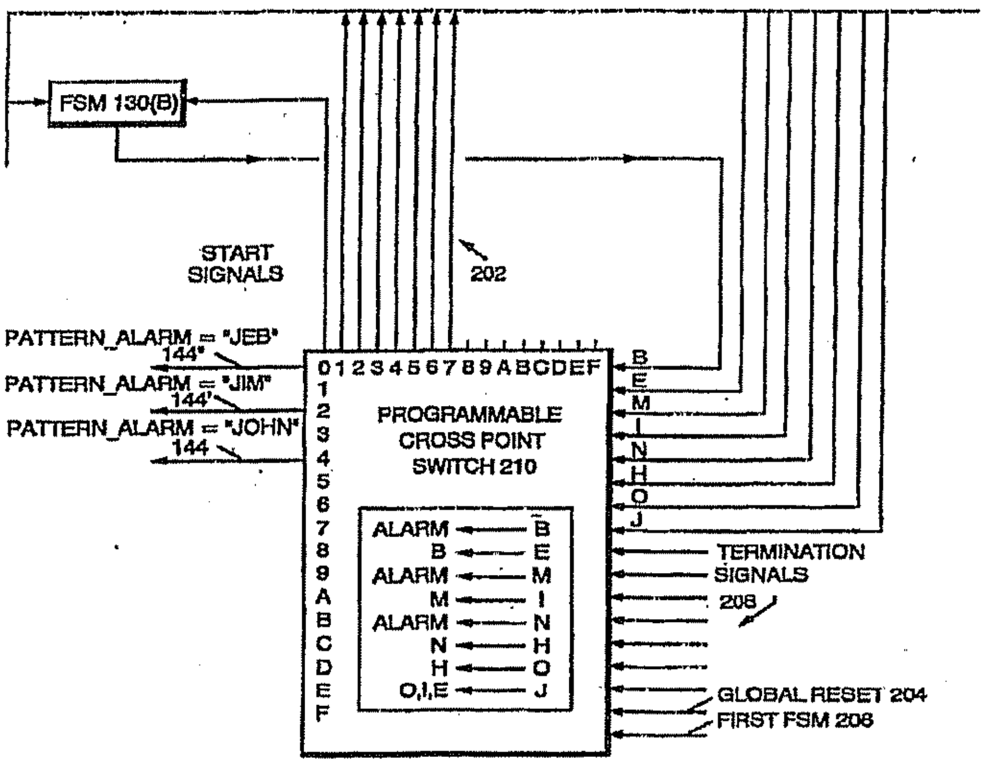
45 Claims, 23 Drawing Sheets


RESPONDER 300
144
144'
144"
PATTERN ALARM SIGNAL
PATTERN ALARM ENCODER 301
REGISTERS 303
SECURITY AGENT IDENTIFIER 321
SECURITY CODE 323
SEQUENCE NUMBER (COUNTER) 325
CRYPTOGRAPHIC KEY 327
PROGRAM LATCH 302
DATA ENCRYPTION ALGORITHM 304
SECURITY ALERT MESSAGE (SAM) PRODUCTION MEANS 331
MESSAGE AUTHENTICATION CODE (MAC) PRODUCTION MEANS 333
341
SAM
MAC
342
PROCESSOR 305
SECURITY ALERT MESSAGE TRANSMISSION MEANS 306
BIT STREAM 124

SC188684

FIG. 1A-1


BIT STREAM
124
START
SIGNALS
202
PARALLEL FSM
ADAPTIVE MONITOR
100
TERMINATION
SIGNALS
208
FSM 130(J)
FSM 130(O)
FSM 130(H)
FSM 130(N)
FSM 130(I)
FSM 130(M)
FSM 130(E)



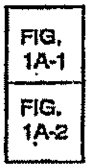
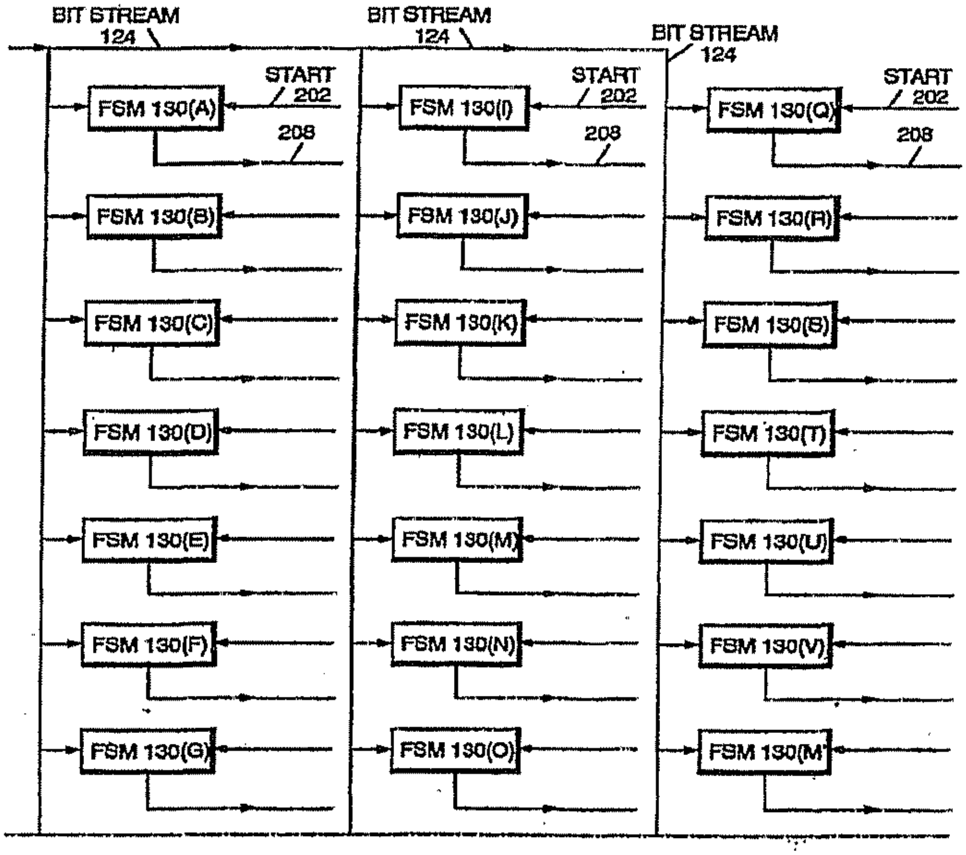


FIG. 1A

FIG. 1A-2

FIG. 1B-1


BIT STREAM
124
BIT STREAM
124
BIT STREAM
124
START
202
START
202
START
202
208
208
208
FSM 130(A)
FSM 130(B)
FSM 130(C)
FSM 130(D)
FSM 130(E)
FSM 130(F)
FSM 130(G)
FSM 130(I)
FSM 130(J)
FSM 130(K)
FSM 130(L)
FSM 130(M)
FSM 130(N)
FSM 130(O)
FSM 130(Q)
FSM 130(R)
FSM 130(S)
FSM 130(T)
FSM 130(U)
FSM 130(V)
FSM 130(W)

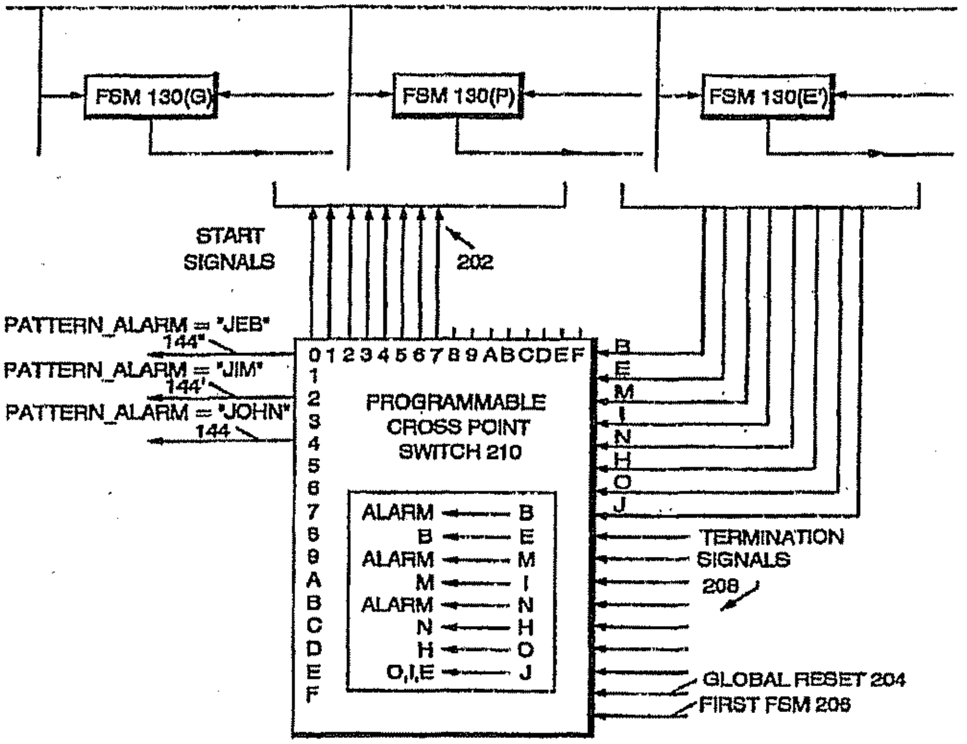
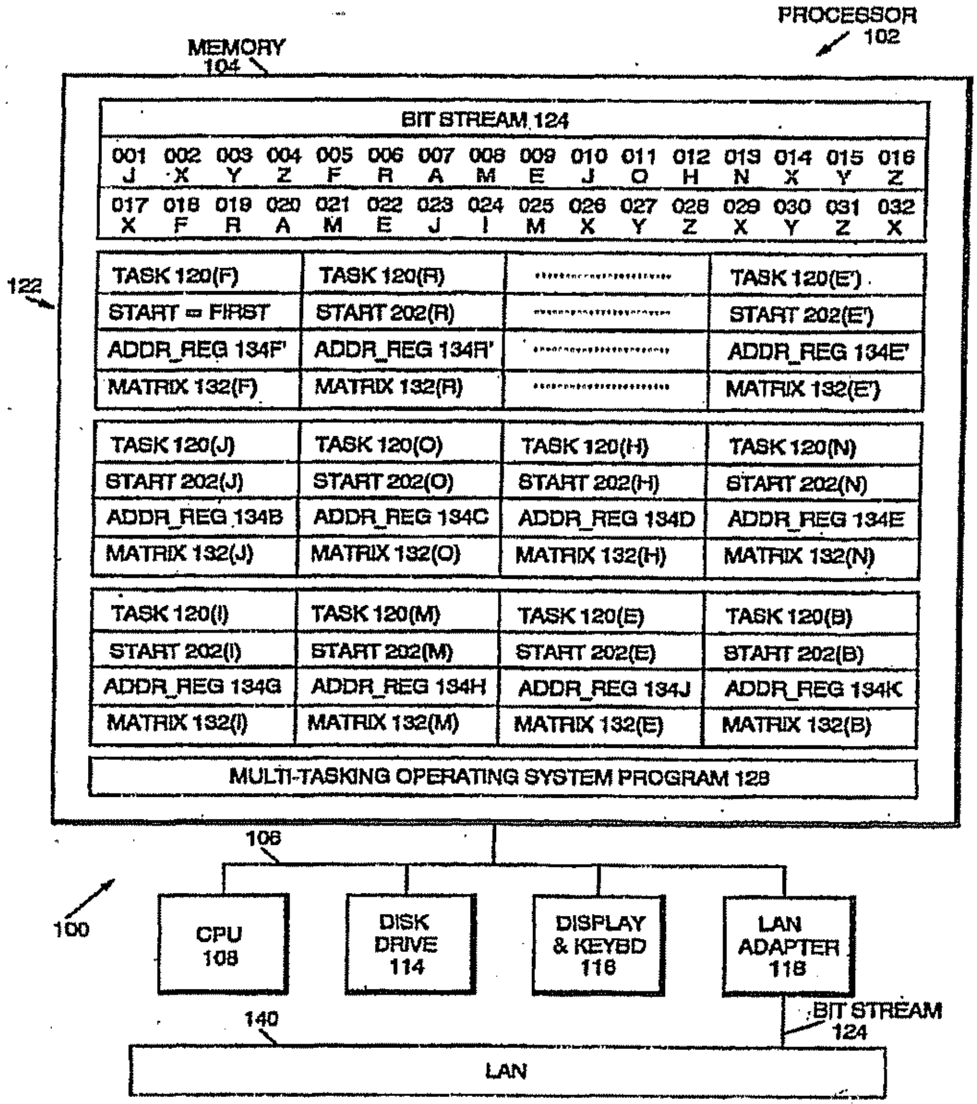


FIG. 1B-2

FIG. 1B

FIG. 1C



BIT STREAM 124

001	002	003	004	005	006	007	008	009	010	011	012	013	014	015	016
J	X	Y	Z	F	R	A	M	E	J	O	H	N	X	Y	Z
017	018	019	020	021	022	023	024	025	026	027	028	029	030	031	032
X	F	R	A	M	E	J	I	M	X	Y	Z	X	Y	Z	X

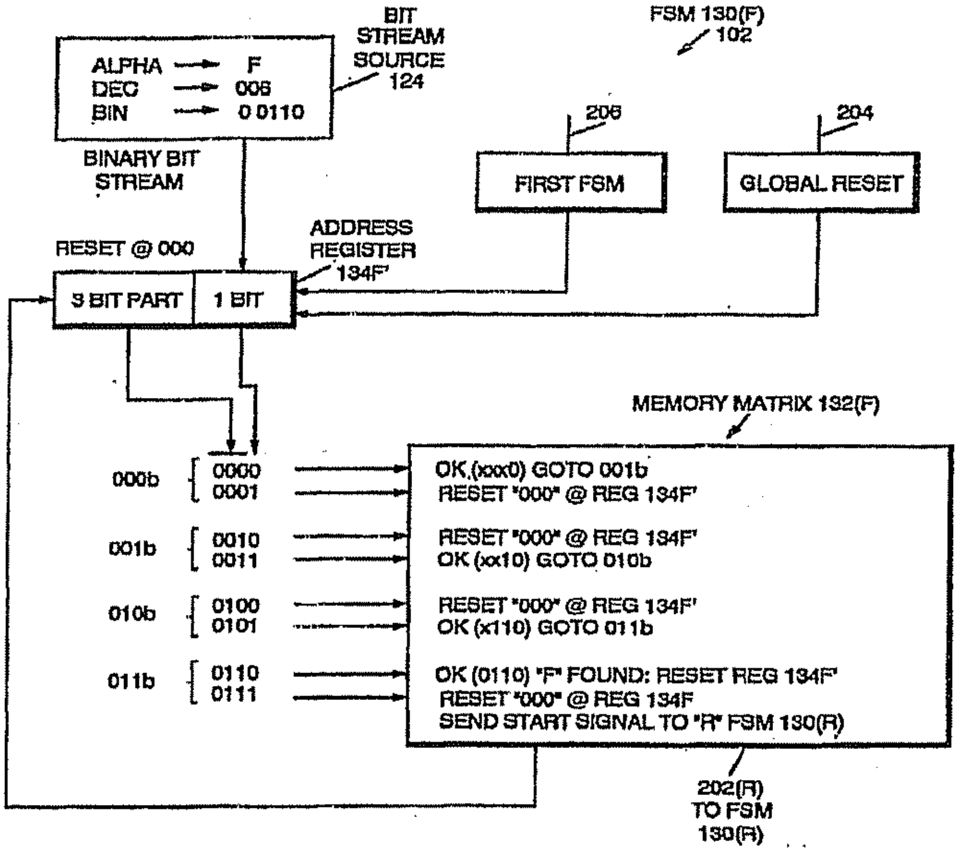
TASK 120(F)	TASK 120(R)	....	TASK 120(E')
START = FIRST	START 202(R)	....	START 202(E')
ADDR_REG 134F'	ADDR_REG 134R'	....	ADDR_REG 134E'
MATRIX 132(F)	MATRIX 132(R)	....	MATRIX 132(E')

TASK 120(J)	TASK 120(O)	TASK 120(H)	TASK 120(N)
START 202(J)	START 202(O)	START 202(H)	START 202(N)
ADDR_REG 134B	ADDR_REG 134C	ADDR_REG 134D	ADDR_REG 134E
MATRIX 132(J)	MATRIX 132(O)	MATRIX 132(H)	MATRIX 132(N)

TASK 120(I)	TASK 120(M)	TASK 120(E)	TASK 120(B)
START 202(I)	START 202(M)	START 202(E)	START 202(B)
ADDR_REG 134G	ADDR_REG 134H	ADDR_REG 134J	ADDR_REG 134K
MATRIX 132(I)	MATRIX 132(M)	MATRIX 132(E)	MATRIX 132(B)

MULTI-TASKING OPERATING SYSTEM PROGRAM 128

FIG. 1D


BIT STREAM SOURCE 124
ALPHA → F
DEC → 006
BIN → 0 0110
FSM 130(F)
102
206
204
FIRST FSM
GLOBAL RESET
BINARY BIT STREAM
RESET @ 000
ADDRESS REGISTER 134F'
3 BIT PART
1 BIT
MEMORY MATRIX 132(F)
000b
0000
0001
001b
0010
0011
010b
0100
0101
011b
0110
0111
OK (xxx0) GOTO 001b
RESET "000" @ REG 134F'
RESET "000" @ REG 134F'
OK (xx10) GOTO 010b
RESET "000" @ REG 134F'
OK (x110) GOTO 011b
OK (0110) "F" FOUND: RESET REG 134F'
RESET "000" @ REG 134F
SEND START SIGNAL TO "R" FSM 130(R)
202(R)
TO FSM 130(R)

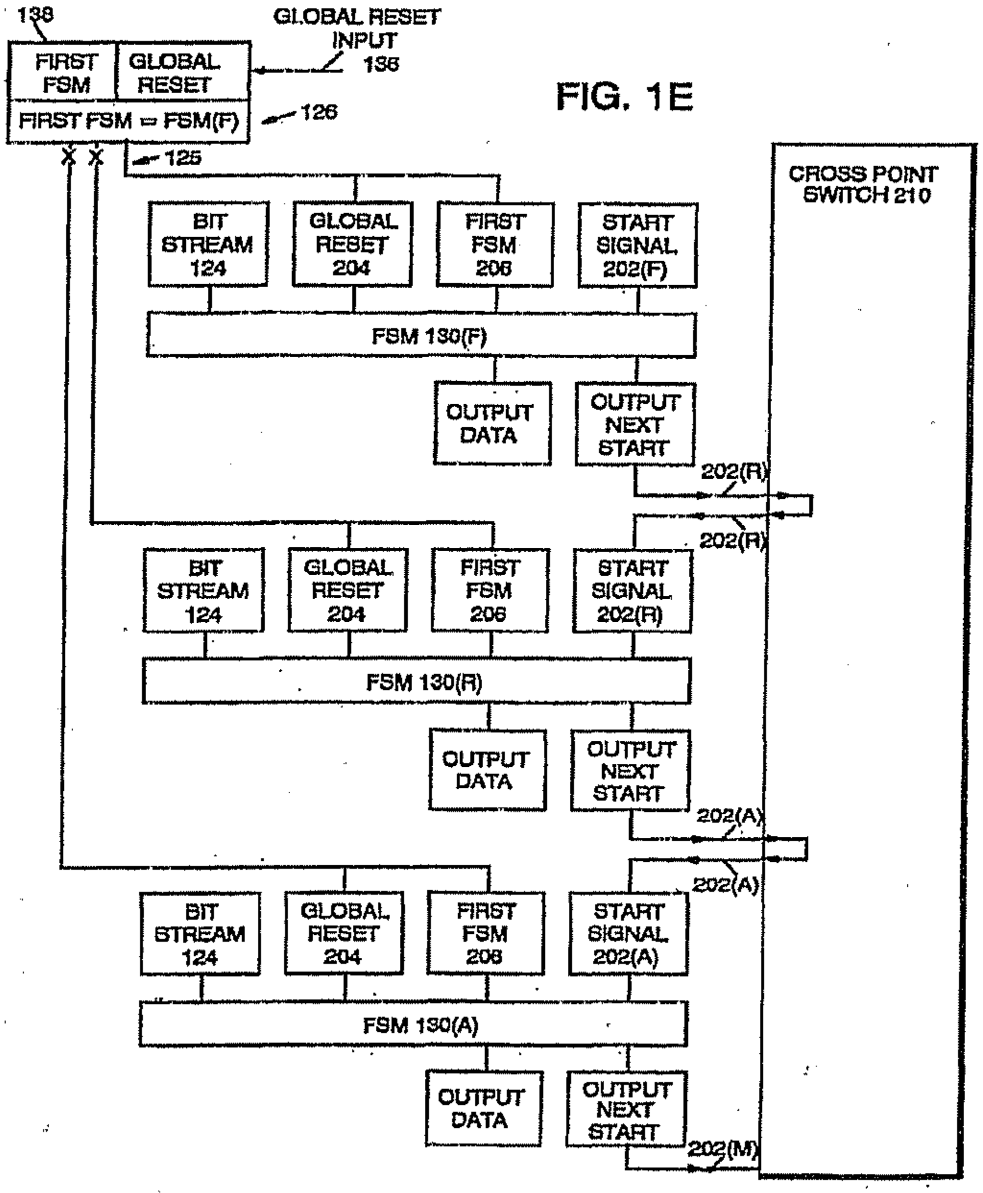

138
GLOBAL RESET INPUT
136
FIRST FSM
GLOBAL RESET
FIRST FSM = FSM(F)
126
125
CROSS POINT SWITCH 210
BIT STREAM 124
GLOBAL RESET 204
FIRST FSM 206
START SIGNAL 202(F)
FSM 130(F)
OUTPUT DATA
OUTPUT NEXT START
202(R)
202(R)
BIT STREAM 124
GLOBAL RESET 204
FIRST FSM 206
START SIGNAL 202(R)
FSM 130(R)
OUTPUT DATA
OUTPUT NEXT START
202(A)
202(A)
BIT STREAM 124
GLOBAL RESET 204
FIRST FSM 206
START SIGNAL 202(A)
FSM 130(A)
OUTPUT DATA
OUTPUT NEXT START
202(M)

FIG. 1E

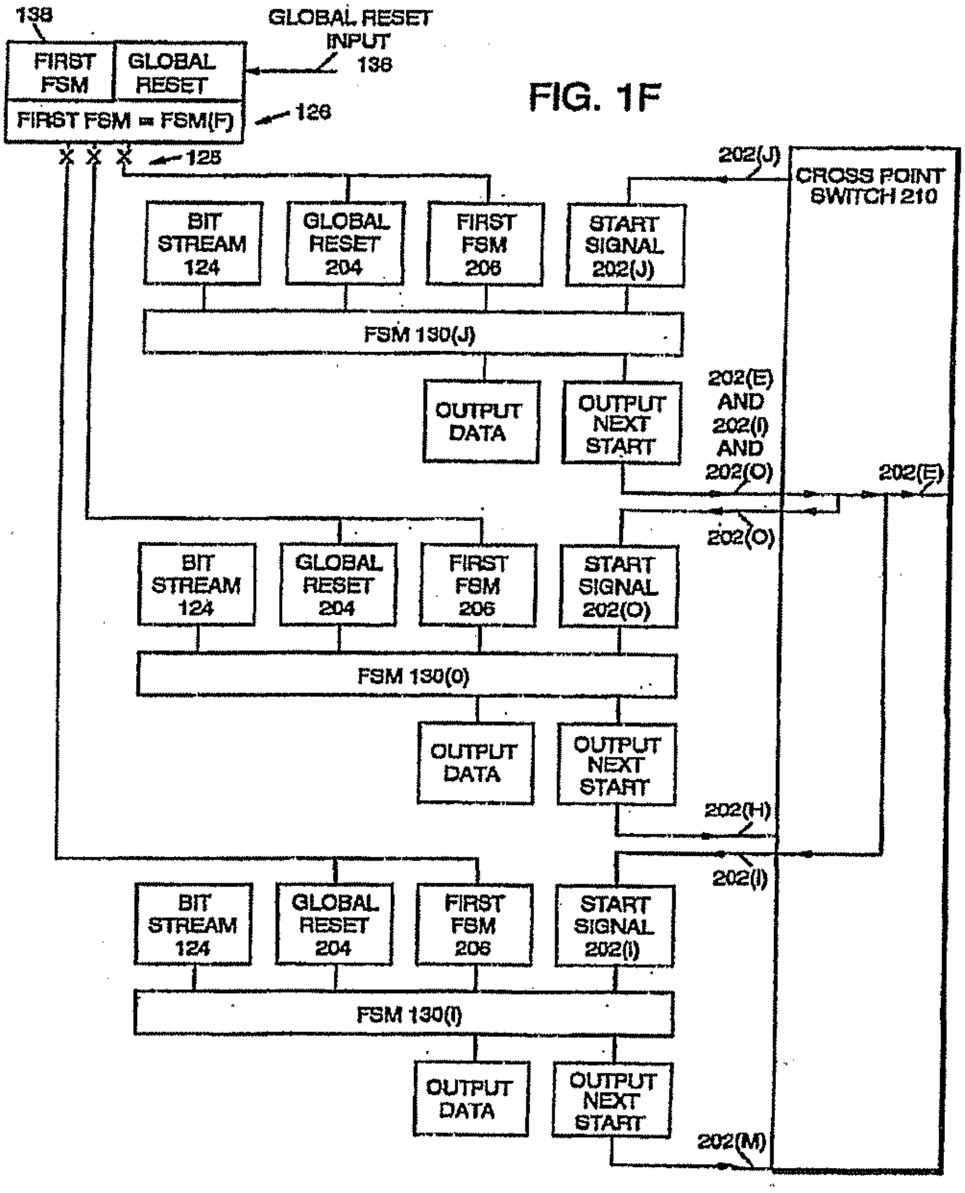

138
GLOBAL RESET INPUT
136
FIRST FSM
GLOBAL RESET
FIRST FSM = FSM(F)
126
128
202(J)
CROSS POINT SWITCH 210
BIT STREAM 124
GLOBAL RESET 204
FIRST FSM 206
START SIGNAL 202(J)
FSM 130(J)
OUTPUT DATA
OUTPUT NEXT START
202(E) AND 202(I) AND 202(O)
202(E)
202(O)
BIT STREAM 124
GLOBAL RESET 204
FIRST FSM 206
START SIGNAL 202(O)
FSM 130(0)
OUTPUT DATA
OUTPUT NEXT START
202(H)
202(I)
BIT STREAM 124
GLOBAL RESET 204
FIRST FSM 206
START SIGNAL 202(I)
FSM 130(I)
OUTPUT DATA
OUTPUT NEXT START
202(M)

FIG. 1F

FIG. 2

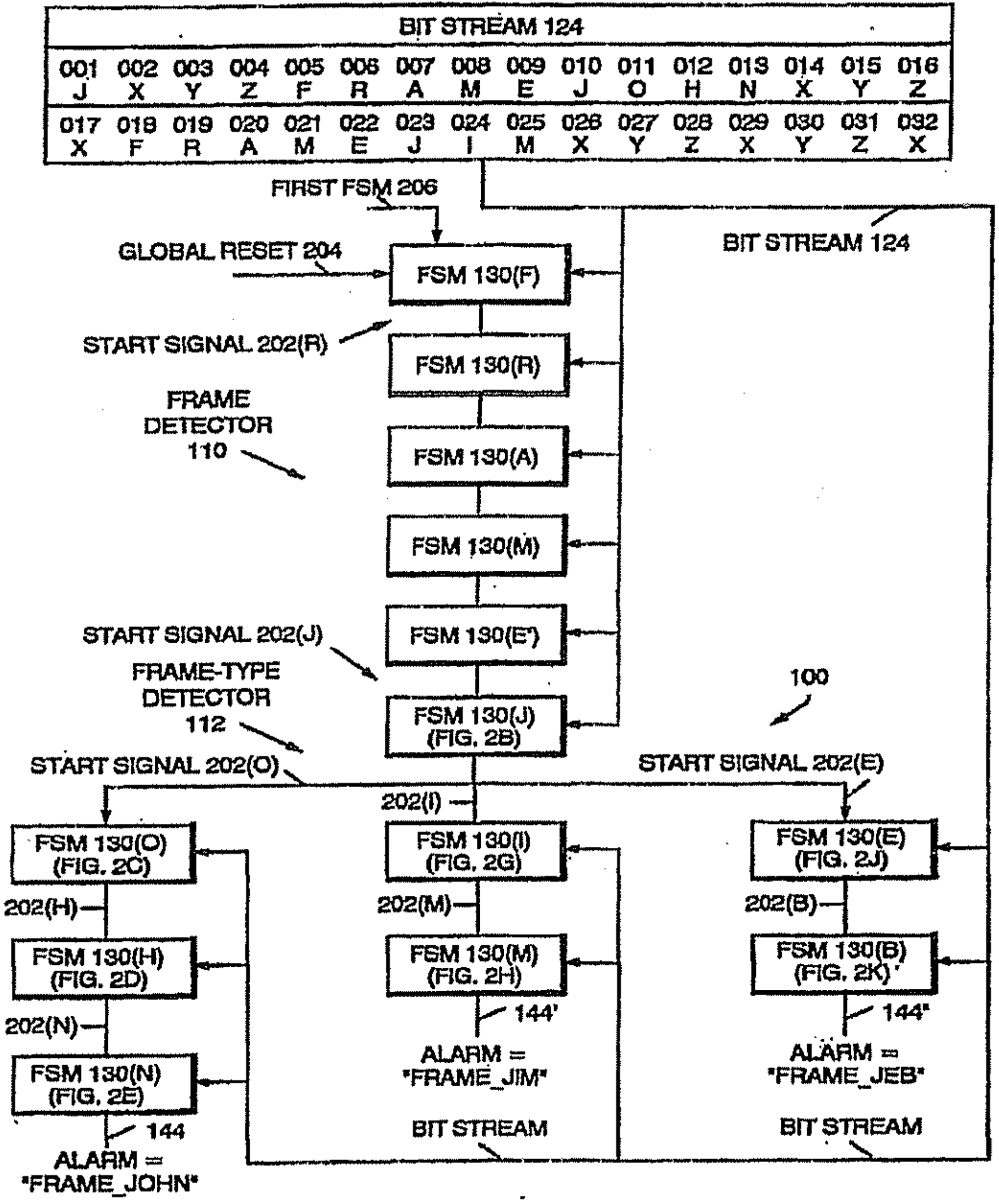

BIT STREAM 124
001 J
002 X
003 Y
004 Z
005 F
006 R
007 A
008 M
009 E
010 J
011 O
012 H
013 N
014 X
015 Y
016 Z
017 X
018 F
019 R
020 A
021 M
022 E
023 J
024 I
025 M
026 X
027 Y
028 Z
029 X
030 Y
031 Z
032 X
FIRST FSM 206
GLOBAL RESET 204
FSM 130(F)
BIT STREAM 124
START SIGNAL 202(R)
FSM 130(R)
FRAME DETECTOR 110
FSM 130(A)
FSM 130(M)
START SIGNAL 202(J)
FSM 130(E')
FRAME-TYPE DETECTOR 112
FSM 130(J) (FIG. 2B)
100
START SIGNAL 202(O)
START SIGNAL 202(E)
202(I)
FSM 130(O) (FIG. 2C)
FSM 130(I) (FIG. 2G)
FSM 130(E) (FIG. 2J)
202(H)
202(M)
202(B)
FSM 130(H) (FIG. 2D)
FSM 130(M) (FIG. 2H)
FSM 130(B) (FIG. 2K)
144'
144"
202(N)
ALARM = "FRAME_JIM"
ALARM = "FRAME_JEB"
FSM 130(N) (FIG. 2E)
144
BIT STREAM
BIT STREAM
ALARM = "FRAME_JOHN"

FIG. 3

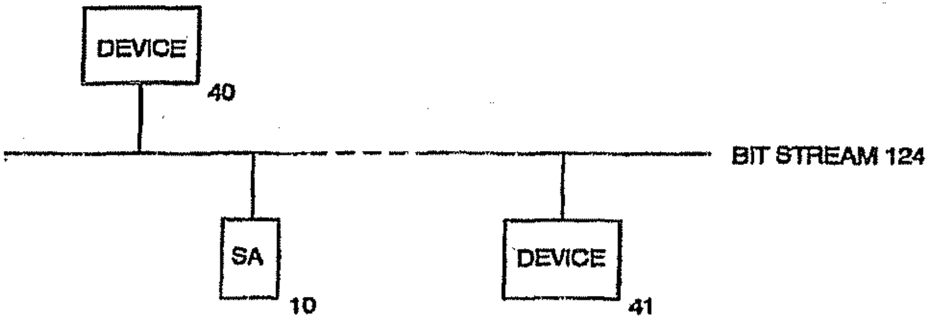

DEVICE
40
BIT STREAM 124
SA
10
DEVICE
41

FIG. 4

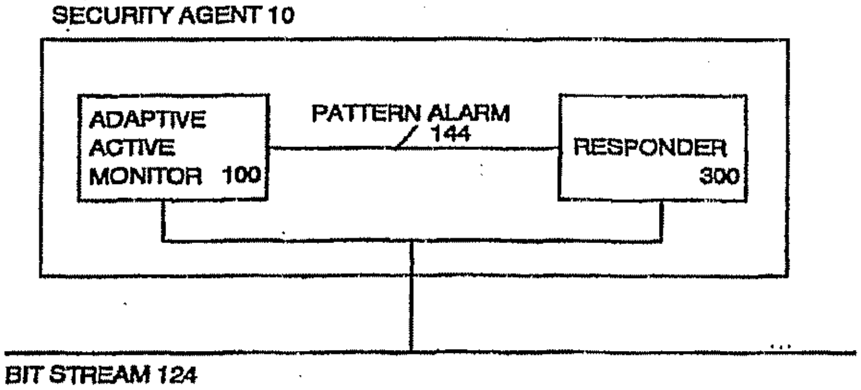

SECURITY AGENT 10
ADAPTIVE ACTIVE MONITOR 100
PATTERN ALARM 144
RESPONDER 300
BIT STREAM 124

FIG. 5

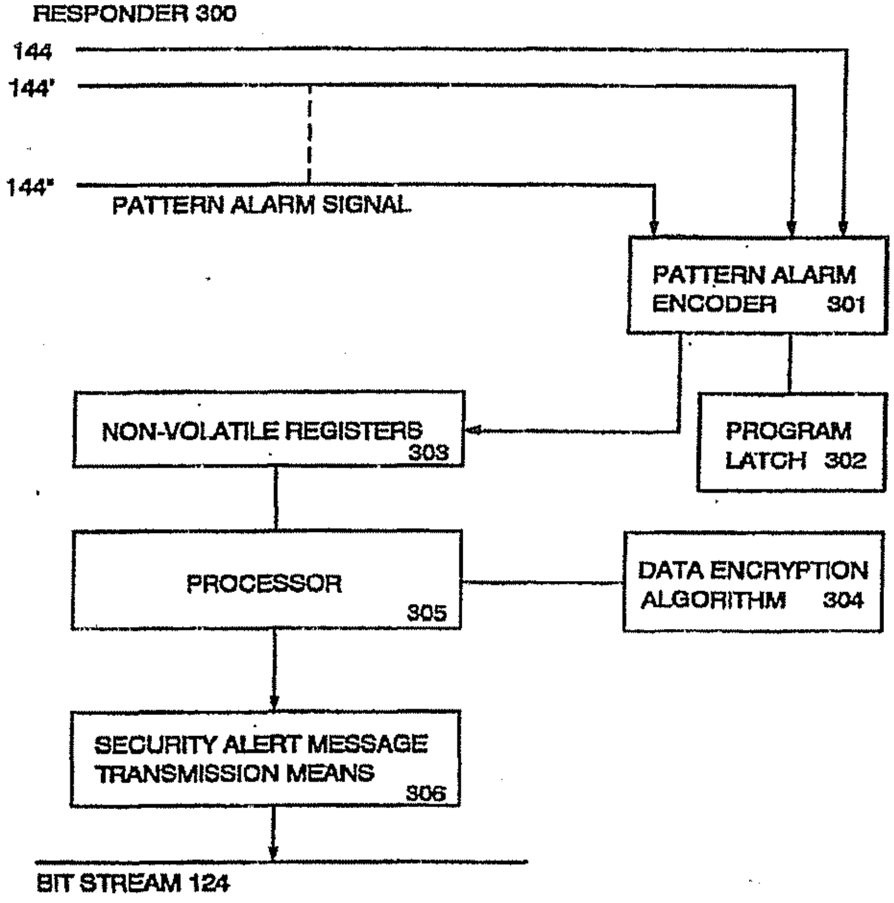
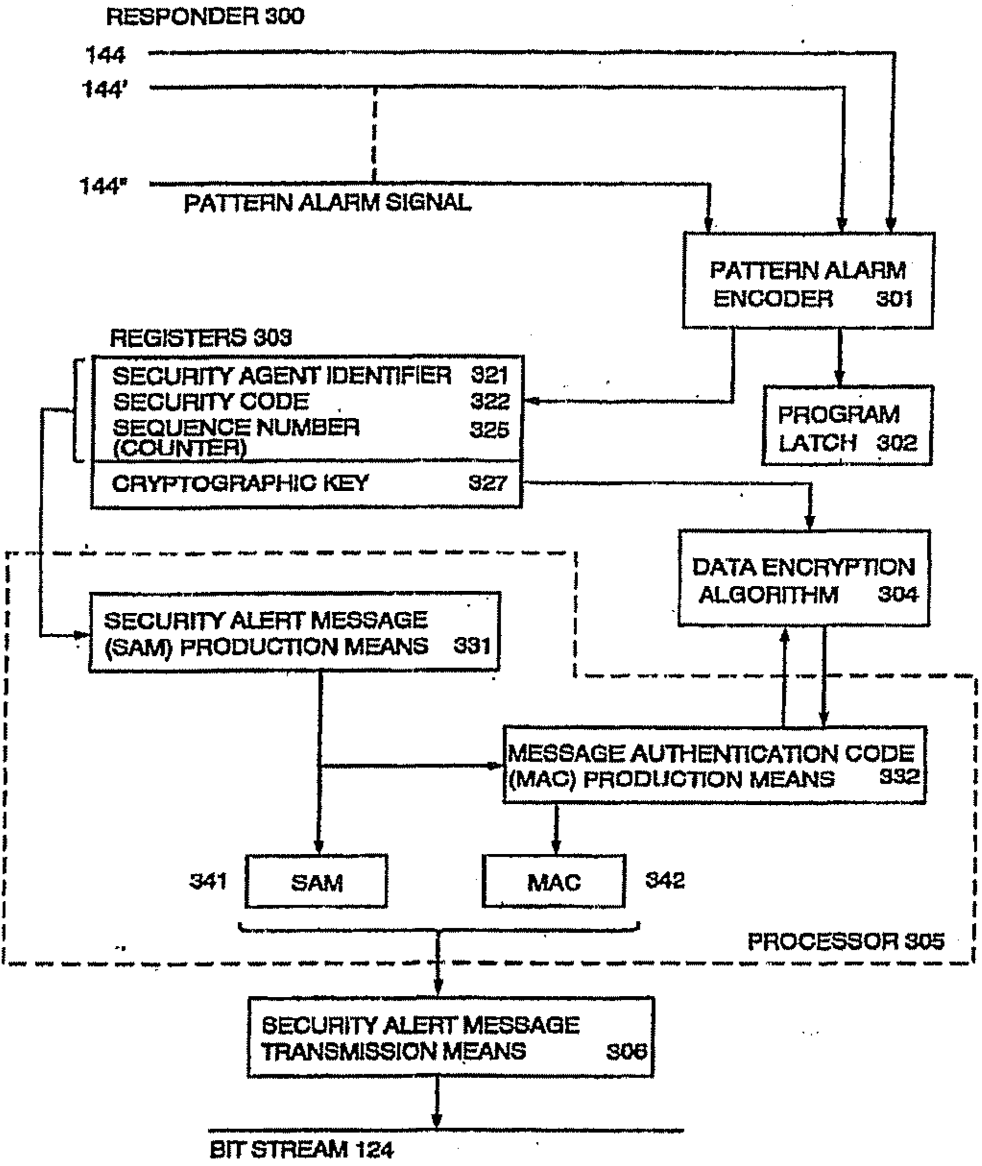

RESPONDER 300
144
144'
144"
PATTERN ALARM SIGNAL
PATTERN ALARM ENCODER 301
NON-VOLATILE REGISTERS 303
PROGRAM LATCH 302
PROCESSOR 305
DATA ENCRYPTION ALGORITHM 304
SECURITY ALERT MESSAGE TRANSMISSION MEANS 306
BIT STREAM 124

FIG. 6


RESPONDER 300
144
144'
144"
PATTERN ALARM SIGNAL
PATTERN ALARM ENCODER 301
REGISTERS 303
SECURITY AGENT IDENTIFIER 321
SECURITY CODE 322
SEQUENCE NUMBER (COUNTER) 325
CRYPTOGRAPHIC KEY 327
PROGRAM LATCH 302
DATA ENCRYPTION ALGORITHM 304
SECURITY ALERT MESSAGE (SAM) PRODUCTION MEANS 331
MESSAGE AUTHENTICATION CODE (MAC) PRODUCTION MEANS 332
341
SAM
MAC
342
PROCESSOR 305
SECURITY ALERT MESSAGE TRANSMISSION MEANS 306
BIT STREAM 124

SECURITY ALERT MESSAGE 341

SECURITY AGENT IDENTIFIER	321
SECURITY CODE CATEGORY TYPE	322 323 324
SEQUENCE NUMBER COUNTER	325

FIG. 7

EXTENDED SECURITY ALERT MESSAGE 341

SECURITY AGENT IDENTIFIER	321
SECURITY CODE CATEGORY TYPE	322 323 324
SEQUENCE NUMBER COUNTER	325
PATTERN ALARM COUNTER VALUE	326

FIG. 9

EXAMPLE EMBODIMENT OF PATTERN ALARMS AND COUNTERS FOR FIG. 8 CONFIGURED FOR VIRUS DETECTION:

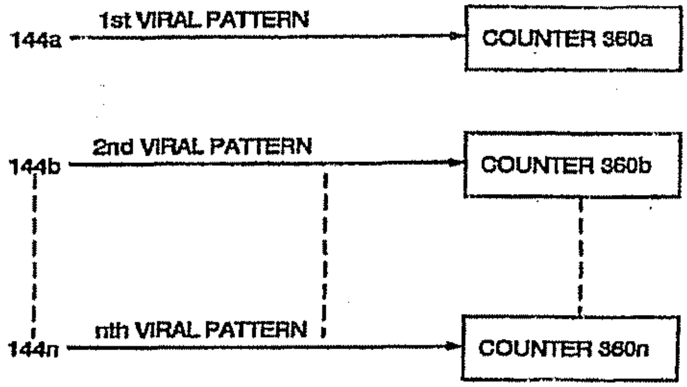


FIG. 12

FIG. 8

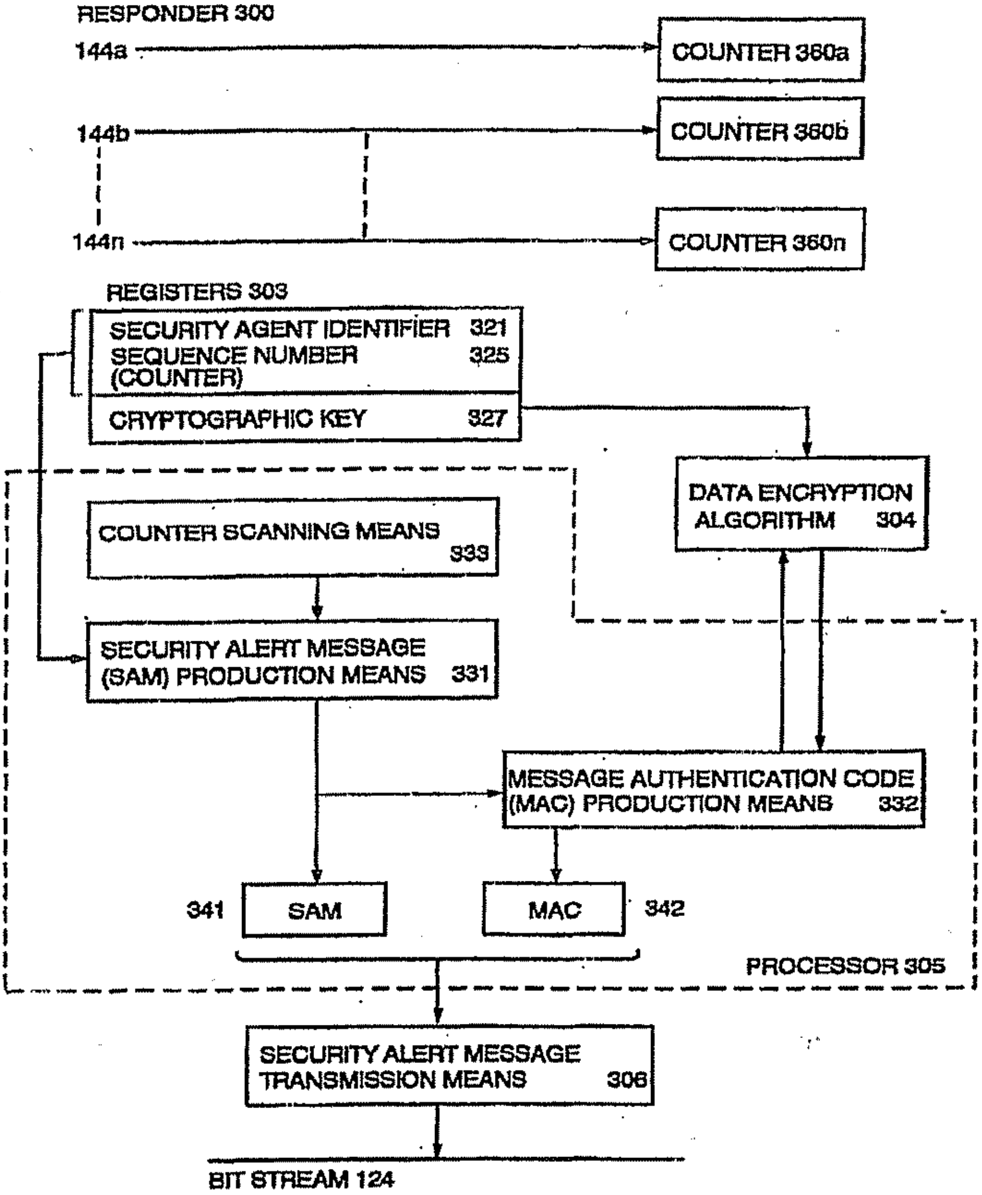
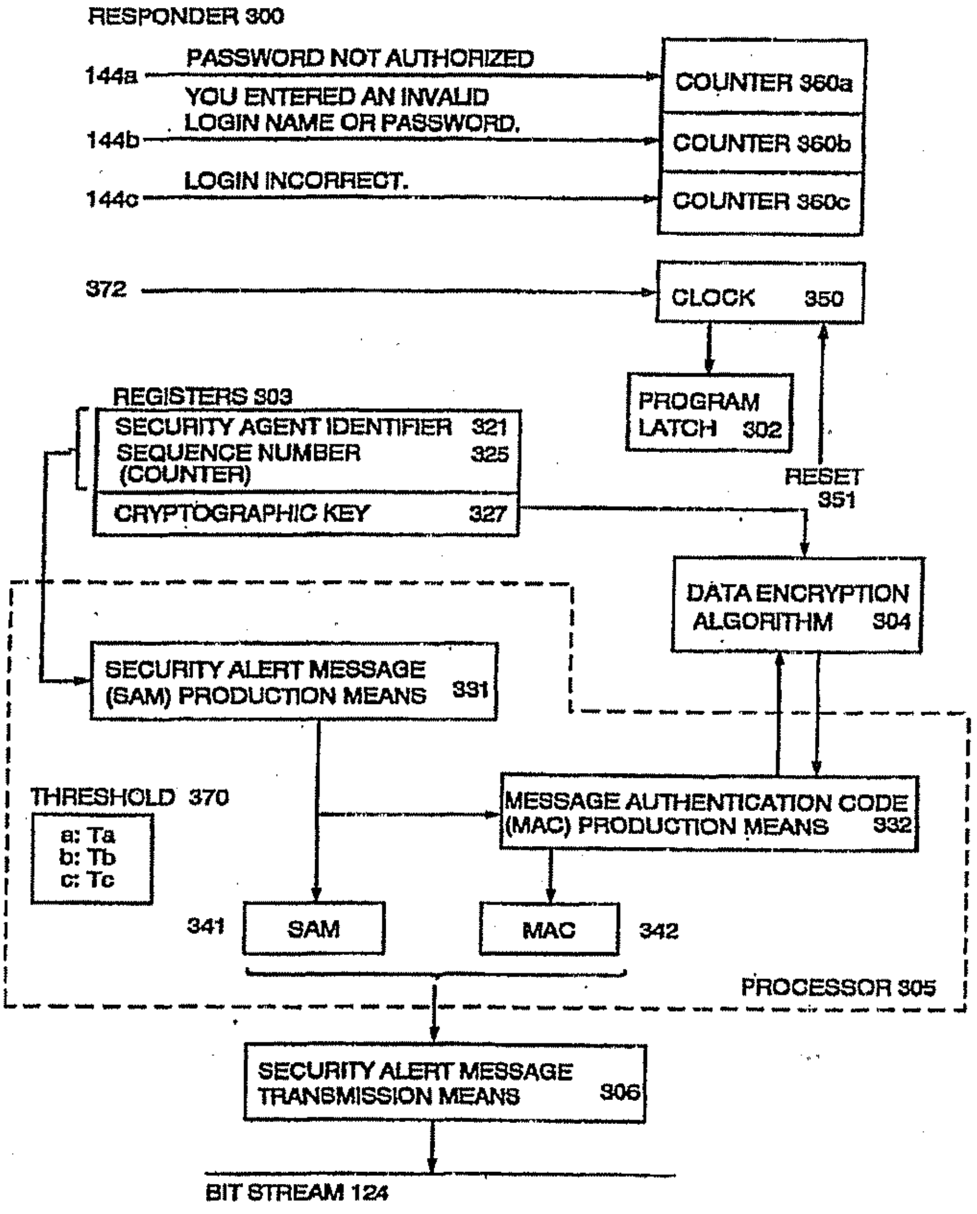

RESPONDER 300
144a
COUNTER 360a
144b
COUNTER 360b
144n
COUNTER 360n
REGISTERS 303
SECURITY AGENT IDENTIFIER 321
SEQUENCE NUMBER (COUNTER) 325
CRYPTOGRAPHIC KEY 327
DATA ENCRYPTION ALGORITHM 304
COUNTER SCANNING MEANS 333
SECURITY ALERT MESSAGE (SAM) PRODUCTION MEANS 331
MESSAGE AUTHENTICATION CODE (MAC) PRODUCTION MEANS 332
341
SAM
MAC
342
PROCESSOR 305
SECURITY ALERT MESSAGE TRANSMISSION MEANS 306
BIT STREAM 124

FIG. 10


RESPONDER 300
144a
PASSWORD NOT AUTHORIZED
COUNTER 360a
YOU ENTERED AN INVALID
LOGIN NAME OR PASSWORD.
144b
COUNTER 360b
LOGIN INCORRECT.
144c
COUNTER 360c
372
CLOCK 350
PROGRAM
LATCH 302
RESET
351
REGISTERS 303
SECURITY AGENT IDENTIFIER 321
SEQUENCE NUMBER 325
(COUNTER)
CRYPTOGRAPHIC KEY 327
DATA ENCRYPTION
ALGORITHM 304
SECURITY ALERT MESSAGE
(SAM) PRODUCTION MEANS 331
THRESHOLD 370
a: Ta
b: Tb
c: Tc
MESSAGE AUTHENTICATION CODE
(MAC) PRODUCTION MEANS 332
341
SAM
MAC
342
PROCESSOR 305
SECURITY ALERT MESSAGE
TRANSMISSION MEANS 306
BIT STREAM 124

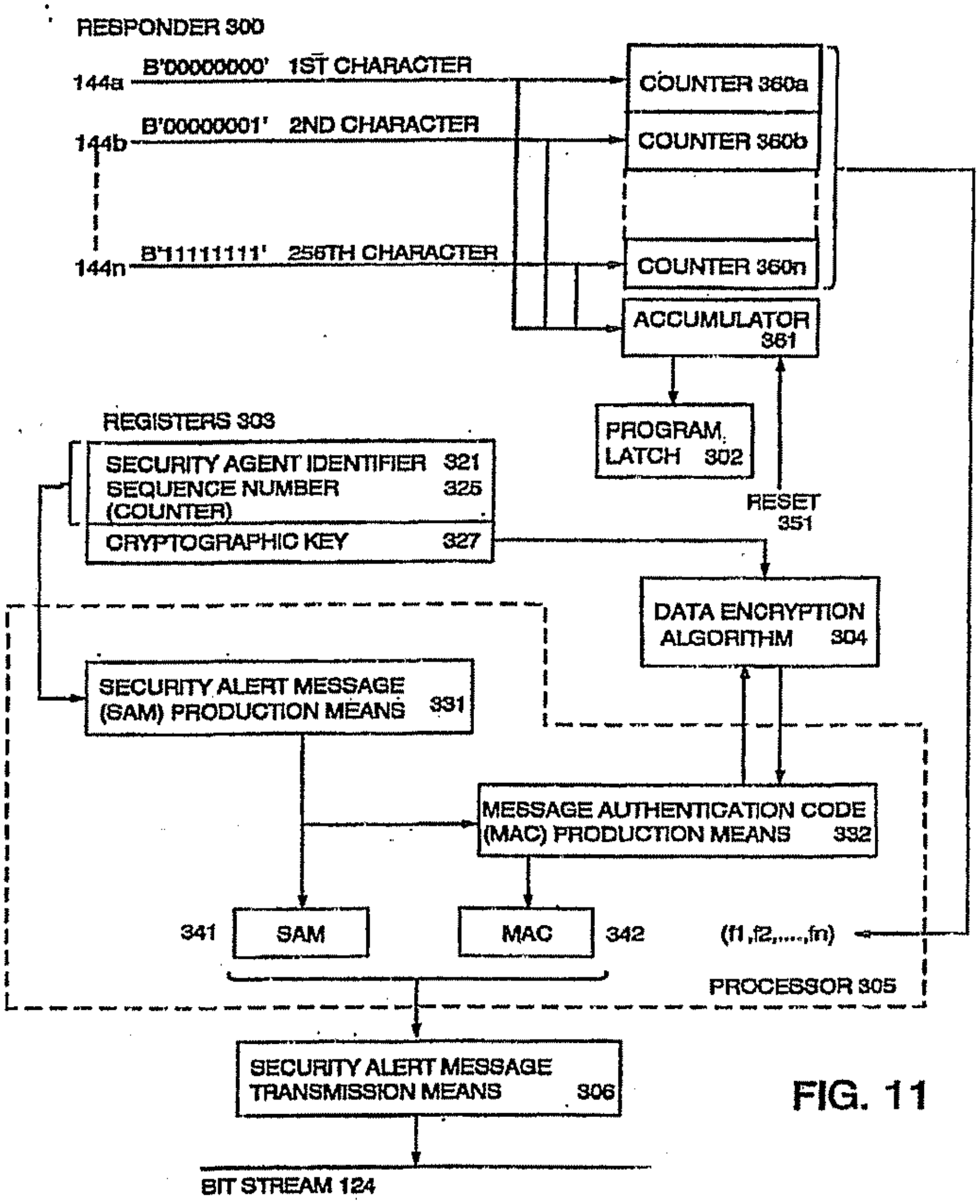

RESPONDER 300
144a
B'00000000' 1ST CHARACTER
144b
B'00000001' 2ND CHARACTER
144n
B'11111111' 256TH CHARACTER
COUNTER 360a
COUNTER 360b
COUNTER 360n
ACCUMULATOR 361
PROGRAM LATCH 302
RESET 351
REGISTERS 303
SECURITY AGENT IDENTIFIER 321
SEQUENCE NUMBER (COUNTER) 325
CRYPTOGRAPHIC KEY 327
DATA ENCRYPTION ALGORITHM 304
SECURITY ALERT MESSAGE (SAM) PRODUCTION MEANS 331
MESSAGE AUTHENTICATION CODE (MAC) PRODUCTION MEANS 332
341
SAM
MAC
342
(f1,f2,....,fn)
PROCESSOR 305
SECURITY ALERT MESSAGE TRANSMISSION MEANS 306
BIT STREAM 124

FIG. 11

FIG. 13

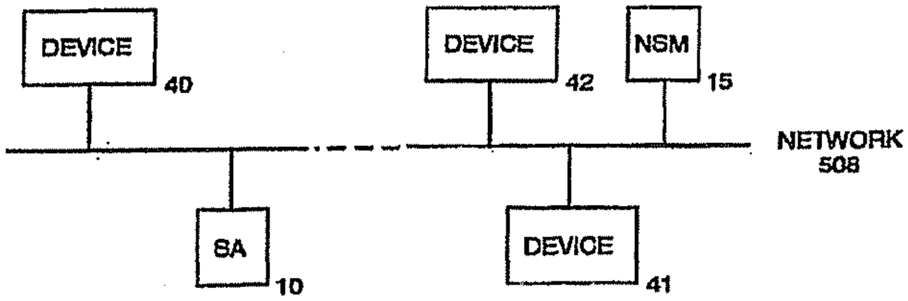

DEVICE
40
DEVICE
42
NSM
15
NETWORK
508
SA
10
DEVICE
41

FIG. 14

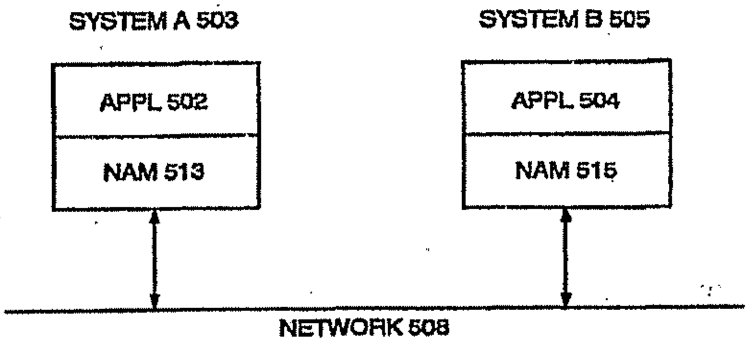

SYSTEM A 503
APPL 502
NAM 513
SYSTEM B 505
APPL 504
NAM 515
NETWORK 508

FIG. 15

LAYER:	FUNCTION:
LAYER 7 APPLICATION LAYER	* PROVIDES ACCESS TO THE COMMUNICATIONS ENVIRONMENT FOR APPLICATIONS AND USERS
LAYER 6 PRESENTATION LAYER	* PROVIDES TRANSLATION OF DATA REPRESENTATION (SYNTAX), E.G., ASCII TO EBCDIC
LAYER 5 SESSION LAYER	* ESTABLISHES, MANAGES, AND TERMINATES SESSIONS BETWEEN COOPERATING, COMMUNICATING APPLICATIONS
LAYER 4 TRANSPORT LAYER	* PROVIDES RELIABLE, ERROR-FREE END-TO-END TRANSFER OF DATA INCLUDING ERROR RECOVERY AND FLOW CONTROL
LAYER 3 NETWORK LAYER	* PROVIDES DATA SWITCHING AND ROUTING CAPABILITIES TO ESTABLISH AND MAINTAIN END-TO-END CONNECTIVITY
LAYER 2 DATA LINK LAYER	* PROVIDES FOR THE RELIABLE TRANSFER OF DATA FROM POINT-TO-POINT, I.E., ACROSS A SINGLE PHYSICAL LINK
LAYER 1 PHYSICAL LAYER	* CONCERNED WITH THE ELECTRICAL TRANSMISSION OF THE BIT STREAM OVER THE NETWORK MEDIA

FIG. 16

LAYER:	FUNCTION:
TCP/UDP TRANSPORT LAYER	* PROVIDES RELIABLE, ERROR-FREE END-TO-END TRANSFER OF DATA INCLUDING ERROR RECOVERY AND FLOW CONTROL; TCP IS CONNECTION-ORIENTED AND UDP IS CONNECTIONLESS
IP NETWORK LAYER	* PROVIDES DATA SWITCHING AND ROUTING CAPABILITIES TO ESTABLISH AND MAINTAIN END-TO-END CONNECTIVITY
LAYER 2 DATA LINK LAYER	* PROVIDES FOR THE RELIABLE TRANSFER OF DATA FROM POINT-TO-POINT, I.E., ACROSS A SINGLE PHYSICAL LINK
LAYER 1 PHYSICAL LAYER	* CONCERNED WITH THE ELECTRICAL TRANSMISSION OF THE BIT STREAM OVER THE NETWORK MEDIA

FIG. 17

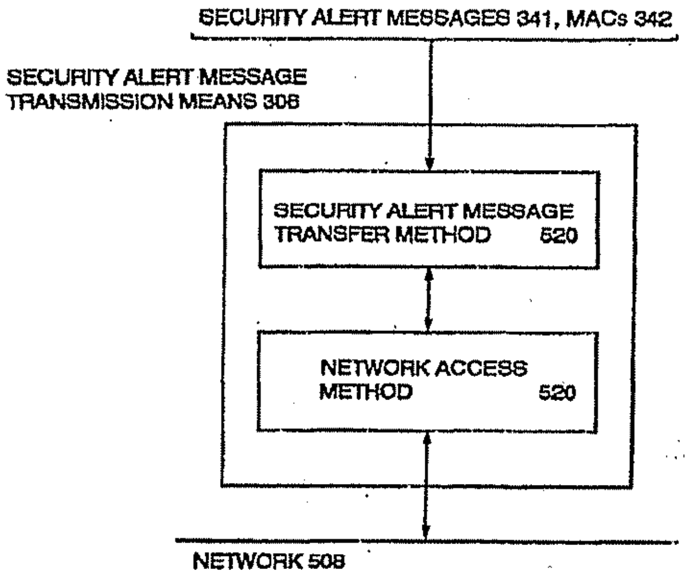
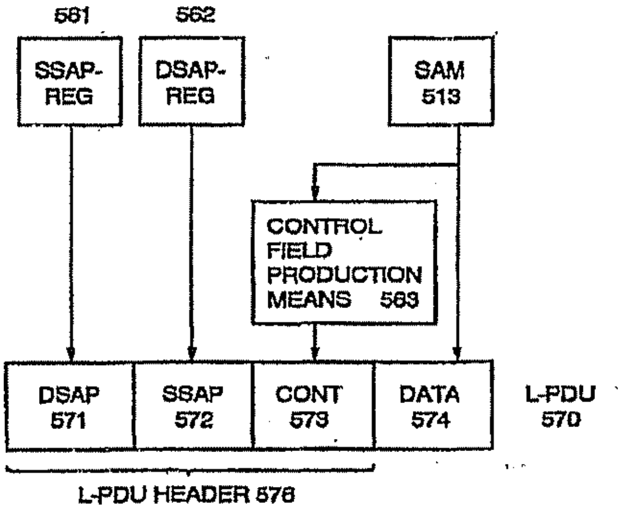

SECURITY ALERT MESSAGES 341, MACs 342
SECURITY ALERT MESSAGE TRANSMISSION MEANS 306
SECURITY ALERT MESSAGE TRANSFER METHOD 520
NETWORK ACCESS METHOD 520
NETWORK 508

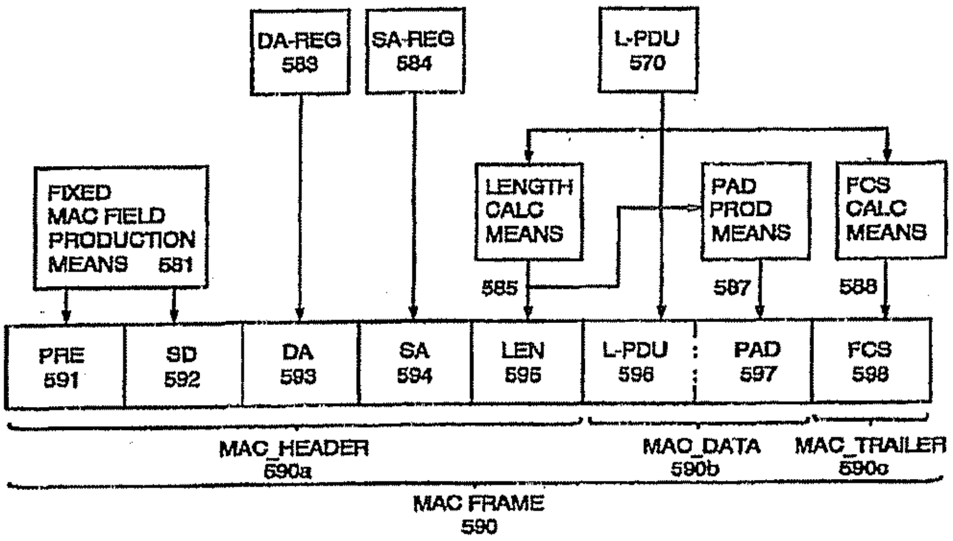
FIG. 18

LAYER:	FUNCTION:
LAYER 2 LOGICAL LINK CONTROL	* PROVIDES MEDIA-INDEPENDENT MEANS FOR POINT-TO-POINT TRANSFER OF DATA (IEEE 802.2)
LAYER 2 MEDIA ACCESS CONTROL	* PROVIDES MEDIA-DEPENDENT FUNCTIONS TO TRANSFER DATA FROM POINT-TO-POINT IN A CSMA/CD LAN (IEEE 802.3)
LAYER 1 PHYSICAL LAYER	* CONCERNED WITH THE ELECTRICAL TRANSMISSION OF THE BIT STREAM OVER THE NETWORK MEDIA (E.G., MANCHESTER ENCODING)

FIG. 19



KEY:
DSAP - DESTINATION SERVICE ACCESS POINT
SSAP - SOURCE SERVICE ACCESS POINT
CONT - CONTROL



KEY:			
	PRE	-	PREAMBLE
	SD	-	STARTING DELIMITER
	DA	-	DESTINATION (MAC) ADDRESS
	SA	-	SOURCE (MAC) ADDRESS
	LEN	-	LENGTH
	L-PDU	-	LLC PDU FROM LLC LAYER
	PAD	-	PADDING (OPTIONAL)
	FCS	-	FRAME CHECK SEQUENCE

FIG. 20

FIG. 21

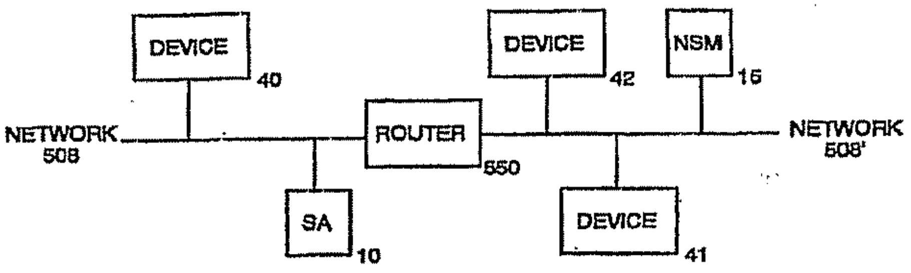


FIG. 22

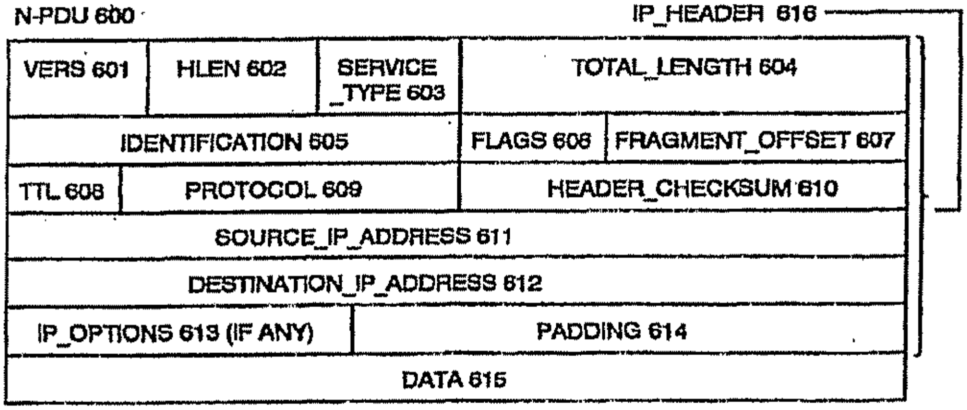

N-PDU 600
IP_HEADER 616
VERS 601
HLEN 602
SERVICE _TYPE 603
TOTAL_LENGTH 604
IDENTIFICATION 605
FLAGS 606
FRAGMENT_OFFSET 607
TTL 608
PROTOCOL 609
HEADER_CHECKSUM 610
SOURCE_IP_ADDRESS 611
DESTINATION_IP_ADDRESS 612
IP_OPTIONS 613 (IF ANY)
PADDING 614
DATA 615

FIG. 23

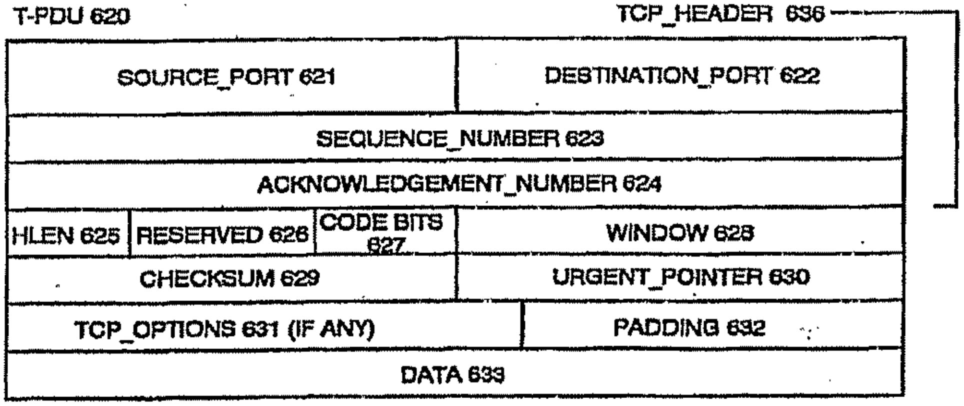
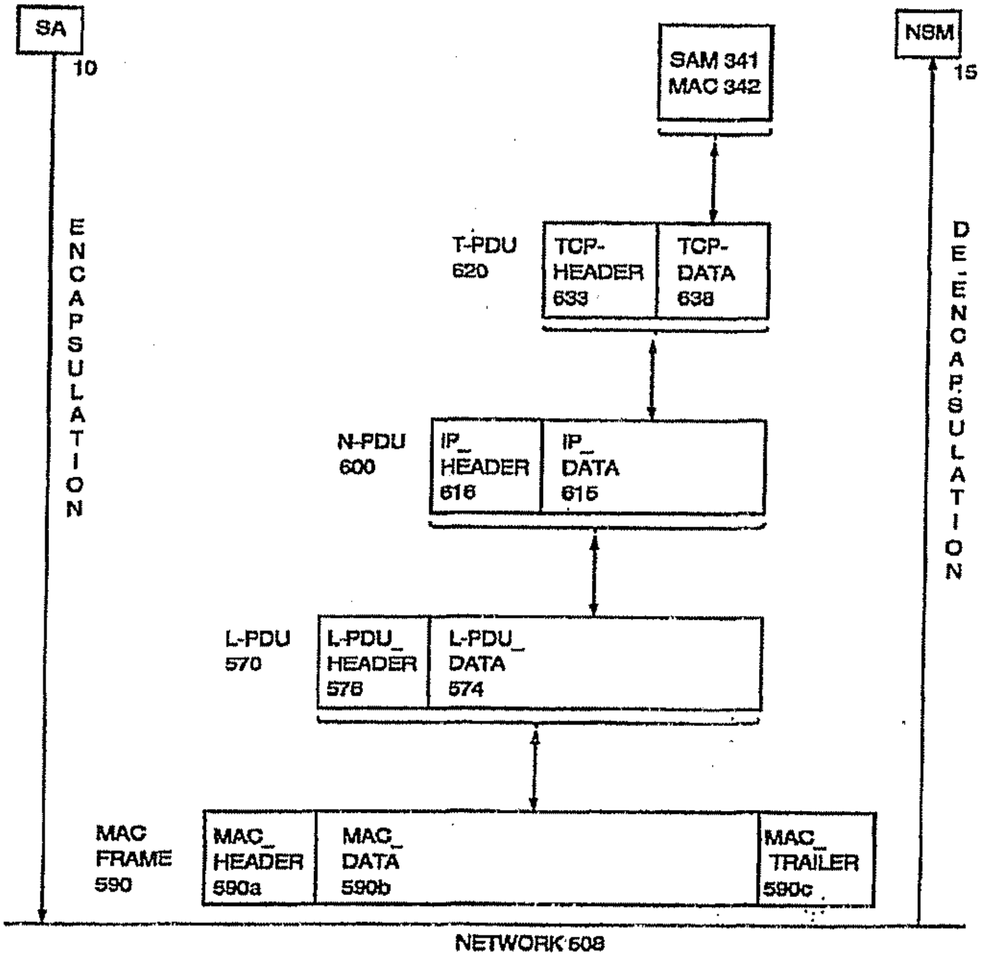

T-PDU 620
TCP_HEADER 636
SOURCE_PORT 621
DESTINATION_PORT 622
SEQUENCE_NUMBER 623
ACKNOWLEDGEMENT_NUMBER 624
HLEN 625
RESERVED 626
CODE BITS 627
WINDOW 628
CHECKSUM 629
URGENT_POINTER 630
TCP_OPTIONS 631 (IF ANY)
PADDING 632
DATA 633

FIG. 24


SA
10
ENCAPSULATION
SAM 341
MAC 342
NSM
15
DE-ENCAPSULATION
T-PDU
620
TCP-
HEADER
633
TCP-
DATA
638
N-PDU
600
IP_
HEADER
616
IP_
DATA
615
L-PDU
570
L-PDU_
HEADER
576
L-PDU_
DATA
574
MAC
FRAME
590
MAC_
HEADER
590a
MAC_
DATA
590b
MAC_
TRAILER
590c
NETWORK 608

NETWORK SECURITY SYSTEM AND METHOD USING A PARALLEL FINITE STATE MACHINE ADAPTIVE ACTIVE MONITOR AND RESPONDER

BACKGROUND OF THE INVENTION

1. Technical Field

The invention disclosed broadly relates to data processing systems and methods and more particularly relates to systems and methods for finite state machine processing in a multimedia data communications environment and to systems and methods for use in a data processing system to enhance network security.

2. Related Patents and Patent Applications

This patent application is related to the copending U.S. patent application Ser. No. 08/024,572, filed Mar. 1, 1993, entitled "Information Collection Architecture and Method for a Data Communications Network," by J. G. Waclawsky, et al., assigned to the IBM Corporation and incorporated herein by reference.

This patent application is also related to the copending U.S. patent application, Ser. No. 08/024,575, filed Mar. 1, 1993, entitled "Event Driven Interface for a System for Monitoring and Controlling a Data Communications Network," by P. C. Hershey, et al., assigned to the IBM Corporation and incorporated herein by reference.

This patent application is also related to the copending U.S. patent application, Ser. No. 08/024,542, filed Mar. 1, 1993, entitled "System and Method for Configuring an Event Driven Interface and Analyzing Its Output for Monitoring and Controlling a Data Communications Network," by J. G. Waclawsky, et al., assigned to the IBM Corporation and incorporated herein by reference.

This patent application is also related to the copending U.S. patent application, Ser. No. 08/138,045, filed Oct. 15, 1993, entitled "System and Method for Adaptive, Active Monitoring of a Serial Data Stream Having a Characteristic Pattern," by P. C. Hershey, et al., assigned to the IBM Corporation and incorporated herein by reference.

This patent application is also related to U.S. Pat. No. 4,918,728, issued Apr. 17, 1988 entitled "Data Cryptography Operations Using Control Vectors" by S. M. Matyas, et al., assigned to the IBM Corporation and incorporated herein by reference.

This patent application is also related to the copending U.S. patent application, Ser. No. 08/004,872, filed Jan. 19, 1993, entitled "An Automatic Immune System for Computers and Computer Networks," by W. C. Arnold, et al., assigned to the IBM Corporation and incorporated herein by reference.

This patent application is also related to copending U.S. patent application, Ser. No. 08/004,871, filed Jan. 19, 1993, entitled "Methods and Apparatus for Evaluating and Extracting Signatures of Computer Viruses and Other Undesirable Software Entities," by J. O. Kephart, assigned to the IBM Corporation and incorporated herein by reference.

3. Background Art

Network security is largely concerned with (1) protecting information from unauthorized disclosure (i.e., information security), (2) protecting information from unauthorized modification or destruction (i.e., information integrity), and (3) ensuring the reliable operation of the computing and networking resources. Cryptography is often used to protect the secrecy and integrity of

stored and transmitted data. Ensuring the reliable operation of computing and networking resources is fundamentally a harder problem to solve—one must ensure the functional correctness of the system. The term "reliable operation" means that a system operates correctly, in the way it was intended.

A well-rounded approach to computer and network security will balance the use of cryptographic techniques—for protecting the secrecy and integrity of data—and monitoring techniques—for detecting anomalous network conditions (security events) that may signal the presence of an intruder or intruder agent. A network security administrator, so-notified of a potential intruder or intruder agent, may take one or more possible actions in response to such a notification. Provided that the response is timely, the possible harmful effects of an intruder or intruder agent may be prevented or minimized or localized. Detecting a problem, or potential problem, seems to be the first step in coping with the problem.

Today's high-speed (gigabit per second) multimedia networks consisting of WANs (wide area networks) and LANs (local area networks) can be thought of as a single computing resource comprised of many smaller computing resources spread over a large geographical area. The network as a whole provides network-wide services to its users, in a transparent fashion. It must be capable of communicating voice, image, and text, to name a few. In this new environment, the ability to monitor data flowing over a network, and the ability to react to anomalous conditions, in real time through appropriate network-level actions, seems fundamental to the maintenance of reliable network-wide services.

The co-pending patent application by Hershey et al. entitled "System and Method for Adaptive, Active Monitoring of a Serial Data Stream Having a Characteristic Pattern", Ser. No. 08/138,045 describes a programmable method for detecting characteristic data patterns of diverse size transmitted over high-speed data links. Unlike more traditional method in data is sampled and stored in a log, the finite state machine (FSM) information monitoring means of the Hershey invention, cited above, is programmed to "look" for interesting patterns of concern. In this way, the FSM discards most of the high-speed information bits and concentrates only on patterns of interest. In short, the FSM signals a pattern match as opposed to collecting and storing data in a log, which must then be processed by some other network function. The FSM information monitoring means is coupled to the network, and in response to detecting a prescribed pattern, outputs a control signal to the network to alter communication characteristics thereof. How this control signal is handled depends on the application supported by the FSM information monitoring means. The Hershey et al. application, Ser. No. 08/138,045 is limited in its teaching of how systems can respond to the detection of prescribed patterns, concentrating more of the problem of pattern detection itself. Moreover, the Hershey application, referred to above as well as other prior art, does not teach a unified method for (a) monitoring of security events—virus patterns, natural language patterns, and intrusion detection patterns—on high-speed communication links, (b) reporting detected security events to a network security manager, and (c) responding on a network-level to detected security events as a means to thwart, counter, minimize, or isolate their possible harmful effects.

3.1 Virus Detection

A virus is a computer program that (1) propagates itself through a system or network of systems and (2) appears to the user to perform a legitimate function but in fact carries out some illicit function that the user of the program did not intend. See M. Gasser, "Building a Secure Computer System," Van Nostrand Reinhold, N.Y., 1988. A computer virus has been defined by Frederick B. Cohen (A Short Course on Computer Viruses, page 11), as a program that can infect other programs by modifying them to include a, possibly evolved, version of itself. As employed herein, a computer virus is considered to include an executable assemblage of computer instructions or code that is capable of attaching itself to a computer program. The subsequent execution of the viral code may have detrimental effects upon the operation of the computer that hosts the virus. Some viruses have an ability to modify their constituent code, thereby complicating the task of identifying and removing the virus. Another type of undesirable software entity is known as a "Trojan Horse." A Trojan Horse is a block of undesired code that is intentionally hidden within a block of desirable code. A virus is typically identified by a 'signature', i.e., a sequence of data bits sufficient to distinguish the virus from other data, or sufficient to raise a warning flag that a virus may be present. In the latter case, further checking and identification of the candidate virus must be performed.

Viruses can be detected by two primary means: (1) modification detection and (2) pattern detection via a scanner. With modification detection, checksums or cryptographic hash values are used to detect changes in executable codes. These changes are reported to a system manager who then decides whether the change is expected (e.g., due to a recent software upgrade) or unexpected (e.g., due to viral infection or unauthorized modification). This method usually requires manual intervention to add, delete, or modify system files in order to ensure adequate coverage and to limit the number of false alarms. A list of checksums must be maintained for all files to be protected. This method is not practical in a high-speed communications environment for several reasons: (1) the overhead imposed by computing checksums, (2) the unpredictability of data flowing on the communications medium, and (3) the requirement for transporting and storing reference checksums for use in comparing with the computed checksums.

A widely-used method for the detection of computer viruses and other undesirable software entities is known as a scanner. A scanner searches through executable files, boot records, memory, and any other areas that might harbor executable code, for the presence of known undesirable software entities. Typically, a human expert examines a particular undesirable software entity in detail and then uses the acquired information to create a method for detecting it wherever it might occur. In the case of computer viruses, Trojan Horses, and certain other types of undesirable software entities, the detection method that is typically used is to search for the presence of one or more short sequences of bytes, referred to as signatures, which occur in that entity. The signature(s) must be chosen with care such that, when used in conjunction with a suitable scanner, they are highly likely to discover the entity if it is present, but seldom give a false alarm, known as a false positive. The requirement of a low false positive rate amounts to requiring that the signature(s) be unlikely to appear in programs that are normally executed on the computer. Typically, if the entity is in the form of binary machine code, a human expert selects signatures by transforming the binary machine code into a human-readable format, such as assembler code, and then analyzes the human-readable code. In the case where that entity is a computer virus, the expert typically discards portions of the code which have a reasonable likelihood of varying substantially from one instance of the virus to another. Then, the expert selects one or more sections of the entity's code which appear to be unlikely to appear in normal, legitimate programs, and identifies the corresponding bytes in the binary machine code so as to produce the signature(s). The expert may also be influenced in his or her choice by sequences of instructions that appear to be typical of the type of entity in question, be it a computer virus, Trojan horse, or some other type of undesirable software entity.

With pattern detection via a scanner, system files are periodically scanned for patterns, which consist of a set of pre-defined virus "signatures." Pattern matches are reported to the system manager who then decides whether the match represents a misdiagnosis or an actual viral infection. A virus consists of one or more fixed-length signature patterns, so the number of virus signatures is proportional to the number of viruses. A list of virus signatures must be maintained for each virus. The pattern search usually proceeds in a serial fashion, scanning each file one at a time, comparing the records of the file with each signature pattern in turn. This form of pattern detection is not suitable for a high speed communications environment because of the delay caused by the serial, fixed-signature search pattern. In a high speed communications environment, it would be desirable to search for many different signature patterns in parallel.

Currently, a number of commercial computer virus scanners are successful in alerting users to the presence of viruses that are already known. However, scanners may not be able to find computer viruses for which they have not been programmed explicitly. The problem of dealing with new viruses has typically been addressed by distributing updates of scanning programs and/or auxiliary files containing the necessary information about the latest viruses. However, the increasing rate at which new viruses are being written is widening the gap between the number of viruses that exist and the number of viruses that can be detected by an appreciable fraction of computer users. Thus it is becoming increasingly likely that a new virus will become wide-spread before any remedy is generally available.

It has become clear to many people in the industry that methods for automatically recognizing and eradicating previously unknown or unanalyzed viruses must be developed and installed on individual computers and computer networks. There are a number of articles addressing this problem. In an article entitled "Automated Program Analysis for Computer Virus Detection," by W. C. Arnold, et al, IBM Technical Disclosure Bulletin, July 1991, page 415, is directed to the potential behavior of program objects to determine heuristically whether they may contain computer viruses or similar threats.

An article entitled "The SRI IDES Statistical Anomaly Detector," H. S. Javitz and A. Valdes, Proceedings of the 1991 IEEE Computer Society Symposium on

Research in Security and Privacy, pp. 316–326 is directed to a statistical approach to anomaly detection.

An article entitled "Towards a Testbed for Malicious Code Detection," by R. Lo, P. Kerchen, R. Crawford, W. Ho, and J. Crossley, Lawrence Livermore National Lab Report UCRL-JC-105792, 1991, which describes static and dynamic analysis tools which have been shown to be effective against certain types of malicious code. Such an idea represents another form of anomaly detection.

Copending U.S. patent application Ser. No. 08/004,872, filed Jan. 19, 1993, entitled "An Automatic Immune System for Computers and Computer Networks," by W. C. Arnold, et al., cited above in Related Patents and Patent Applications, describes a method by which a computer can detect the presence of and respond automatically to a computer virus or other undesirable software entity. If the computer is connected to others via a network, it can warn its neighbors about that entity and inform them about how to detect it. The invention provides methods and apparatus to automatically detect and extract a signature from an undesirable software entity, such as a computer virus or worm. It further provides methods and apparatus for immunizing a computer system, and also a network of computer system, against a subsequent infection by a previously unknown and undesirable software entity.

Copending U.S. patent application Ser. No. 08/004,871, filed Jan. 19, 1993, entitled "Methods and Apparatus for Evaluating and Extracting Signatures of Computer Viruses and Other Undesirable Software Entities," by J. O. Kephart, cited above in Related Patents and Patent Applications, describes an automatic computer implemented procedure for extracting and evaluating computer virus signatures. It further provides a statistical computer implemented technique for automatically extracting signatures from the machine code of a virus and for evaluating the probable effectiveness of the extracted signatures in identifying a subsequent instance of the virus.

The described methods of virus detection (modification detection and pattern detection) are based on identifying a viral infection in a stored form of the data—after the infection has already taken place. A different, highly parallel method is required in order to detect the transfer of viral agents across a high-speed communications link where one "looks" for a viral pattern as it flashes past a monitor attached to the bit stream. The Hershey, et al. adaptive, active monitor described in copending U.S. patent application, Ser. No. 08/138,045, filed Oct. 15, 1993, entitled "System and Method for Adaptive, Active Monitoring of a Serial Data Stream Having a Characteristic Pattern," cited above in Related Patents and Patent Applications, is particularly well-suited for scanning of virus signatures in a high-speed communications environment. The prior art with respect to virus detection, virus scanning, signature preparation, virus reporting, etc. is well defined and described as pointed out above. However, the prior art does not teach how to construct a virus scanning apparatus well-suited to very high-speed networks, and more particularly how such an apparatus could be constructed for attachment to a bit stream as a singular entity, integrated within a network-attached device or standing alone, whose purpose is to act as a monitoring and responding device for assuring the integrity and security of a high-speed communications network.

3.2 Natural Language Detection

Natural language detection admits a range of applications which include (1) detection of inappropriate word use in a business environment (e.g., 4-letter words), (2) detection of inappropriate discourse within a business environment (e.g., use of company computer resources for conducting personal business), (3) detection of sensitive words such as the words "Company Confidential" that may signal the transmission of clear information that should be encrypted or the words "copyright protected" that may signal a possible violation of copyright law, and (4) detection of clear versus encrypted data that may signal a possible violation of company policy requiring all traffic on a link to be encrypted.

Each of the natural language detection applications has a range of possible actions that may be taken in response to a detected offending pattern.

3.3 Intrusion Detection

One form of intrusion of particular concern to network security is an adversary who attempts to gain access to a system by issuing repeated login requests. In this case, intrusion detection is aimed at detecting a higher-than-normal frequency of login sequences indicating that someone is repeatedly attempting to login by guessing userids and passwords. In this security application, one does not merely detect the presence of a pattern but the presence of a higher-than-normal frequency of patterns.

Most security applications (including virus detection, natural language detection, and intrusion detection) consist of a detection step and a response step. The Hershey FSM information monitoring means described in U.S. pending patent application Ser. No. 08/138,045, cited above, is particularly suited as a pattern detection means in a high-speed communication environment. Yet for the Hershey FSM information monitoring means to be well-suited as a security device in a high-speed communication environment, it must be adapted to search for patterns particular to security applications and it must be extended to provide a capability for responding, in appropriate ways, to detected patterns. Such a security device (hereinafter called a security agent) must provide both an information monitoring function as well as a real-time responding function.

OBJECTS OF THE INVENTION

It is an object of the invention to provide a security agent consisting of a monitoring means and a responding means which can support security applications.

It is another object of the invention to provide a security agent capable of responding to a detected security event by producing and transmitting a security alert message to a network security manager.

It is another object of the invention to provide a security agent capable of responding to a detected security event by taking a network security administration action consisting of transmitting a security alert to other stations on the network.

It is another object of the invention to provide a security agent capable of responding to a detected security event by taking a network security administration action consisting of setting a flag in an existing transmitted protocol frame to indicate a security event has occurred.

It is another object of the invention to provide a security agent capable of detecting the transmission of infected programs and data across a high-speed communications network.

It is another object of the invention to provide a security agent with an information monitoring means that can be re-programmed in case it is necessary to dynamically reconfigure it to provide a capability in real-time to detect the presence of a suspected offending virus.

It is another object of the invention to provide a security agent capable of detecting inappropriate words, detecting use of computing resources for non-authorized uses (e.g., non-business purposes), detecting the presence of copyright protected data, detecting the presence of company confidential information, and detecting the presence of clear data as opposed to encrypted data—any of which may signal a violation of policy.

It is another object of the invention to provide a security agent capable of detecting intrusions by an adversary who attempts to gain access to a system using repeated login requests.

SUMMARY OF THE INVENTION

These and other objects, features and advantages are accomplished by the invention disclosed herein. A system and method are disclosed which provides a security agent, consisting of a monitoring means and a responding means, which responds to a detected security event in a data communications network, by producing and transmitting a security alert message to a network security manager. The alert is a security administration action which includes setting a flag in an existing transmitted protocol frame to indicate a security event has occurred.

The security agent detects the transmission of infected programs and data across a high-speed communications network. The security agent includes an information monitoring means; consisting of adaptive, active monitor using finite state machines, that can be dynamically re-programmed in case it is necessary to dynamically reconfigure it to provide a capability in real-time to detect the presence of a suspected offending virus.

In an alternate embodiment of the invention, the security agent detects inappropriate words, detecting use of computing resources for non-authorized uses (e.g., non-business purposes), detecting the presence of copyright protected data, detecting the presence of company confidential information, and detecting the presence of clear data as opposed to encrypted data—any of which may signal a violation of policy. In another alternate embodiment of the invention, the security agent detects intrusions by an adversary who attempts to gain access to a system using repeated login requests.

DESCRIPTION OF THE FIGURES

These and other objects, features and advantages will be more fully appreciated with reference to the accompanying figures.

FIG. 1A is a functional block diagram of a parallel finite state machine adaptive monitor, in accordance with the invention.

FIG. 1B is a functional block diagram showing a plurality of finite state machines that can be interconnected by means of a cross point switch.

FIG. 1C illustrates a processor implementation of the invention wherein a plurality of finite state machines are embodied.

FIG. 1D illustrate a first finite state machine serving as the first patterns analysis stage in a plurality of finite state machines.

FIG. 1E illustrate the interconnection of a first finite state machine with successor finite state machines, using a cross point switch.

FIG. 1F illustrates the interconnection of a predecessor finite state machine and a plurality of successor, parallel finite state machines using a cross point switch.

FIG. 2 illustrates an example of an array of finite state machines including a predecessor finite state machine 130(J) which starts a plurality of successor finite state machines, in parallel.

FIG. 3 illustrates a security agent (SA) 10 and communicating devices 40 and 41 connected to a bit stream 124.

FIG. 4 illustrates a security agent (SA) 10 connected to a bit stream 124 consisting of an adaptive, active monitor 100 and a responder 300.

FIG. 5 is a block diagram of a responder 300.

FIG. 6 is a block diagram illustration of the Responder 300 that processes security events, which are characterized as one of a plurality of possible pattern alarms (144, 144′, ..., 144″) output by Adaptive, Active Monitor 100 of FIG. 4.

FIG. 7 depicts a security alert message 341 consisting of a security agent identifier 321, a security code 322 and a sequence number counter 325.

FIG. 8 is a block diagram illustration of an alternate embodiment of the invention (as described in FIG. 6) wherein the pattern alarms 144*a*, 144*b*, ..., 144*n* from the Hershey adaptive, active monitor 100 are applied to counters 360*a*, 360*b*, ..., 360*n*, respectively, in order to prevent adaptive, active monitor 100 (see FIG. 4) of FIG. 4 from over-running responder 300.

FIG. 9 depicts an extended security alert message 341 consisting of a security agent identifier 321, a security code 322, a sequence number counter 325; and a pattern alarm counter value 326.

FIG. 10 is a block diagram illustration of another alternate embodiment of the invention wherein the Hershey adaptive, active monitor 100 of FIG. 4 is configured as an intrusion detector and responder 300 is designed to produce and transmit a security alert message whenever the number of detected pattern alarms of a particular type in a given interval of time reaches a prescribed threshold value.

FIG. 11 is a block diagram illustration of another alternate embodiment of the invention wherein the Hershey, et al. adaptive, active monitor 100 of FIG. 4 is configured to detect the transmission of plain or clear text as opposed to ciphertext.

FIG. 12 is an example embodiment of pattern alarms and counters of FIG. 8 configured for virus detection.

FIG. 13 illustrates a general network consisting of a plurality of network-attached devices, a Security Agent (SA), and a Network Security Monitor (NSM).

FIG. 14 illustrates a simple network architecture consisting of a first application in a first network-attached system communicating through a first network access method to a second application in a second network-attached system via a second network access method.

FIG. 15 depicts the 7-layer communications architecture of the Open Systems Interconnection (OSI) model.

FIG. 16 depicts the Transmission Control Protocol-/Internet Protocol (TCB/IP) protocol stack.

FIG. 17 is a block diagram of the Security Alert Message Transmission Means 306

FIG. 18 depicts the Physical layer and the sub-layers of the Data Link Layer in a Carrier-Sense Multiple

Access with Collision Detection or CSMA/CD (IEEE 802.3) Local Area Network architecture.

FIG. 19 is a block diagram of the implemented Logical Link Control sub layer of the Data Link Layer in the Network Access Method of the SA.

FIG. 20 is a block diagram of the implemented Media Access Control sub-layer of the Data Link Layer in the Network Access Method of the SA.

FIG. 21 is a block diagram illustrating a Security Agent 10 in a first network segment communicating through a router internetworking device to a Network Security Manager 15 in a second network segment.

FIG. 22 depicts the format of the Network layer Protocol Data Unit.

FIG. 23 depicts the format of the Transport layer Protocol Data Unit.

FIG. 24 illustrates the symmetric processes of protocol data unit encapsulation at a transmitting Security Agent and protocol unit de-encapsulation at a receiving Network Security Manager.

DISCUSSION OF THE PREFERRED EMBODIMENT

In accordance with the invention, a security agent incorporates both (1) an adaptive, active monitoring means and (2) a responding means. The adaptive, active monitoring means is based on the adaptive, active monitoring means taught in copending patent application by Paul Hershey and John Waclawsky, entitled 'System and Method for Adaptive, Active Monitoring of a Serial Data Stream Having a Characteristic Pattern', Ser. No: 08/138,045 cited above under Related Patents and Patent Applications, assigned to the IBM Corporation and incorporated herein by reference. The responding means, when coupled to the adaptive, active monitoring means, provides the functionality necessary to implement a security agent in a high-speed communication environment.

The uses and features of the adaptive, active monitor can be summarized.

The adaptive, active monitor is useful in detecting characteristic data patterns in messages on a high-speed data network, such as starting delimiters, tokens, various types of frames, and protocol information. Such serial data streams include serial patterns of binary bits, and can also include serial patterns of multiple state symbols, such as the J, K, 0, 1 symbols (four states) of token ring networks. Such serial data streams can further include multiple state symbols in fiber optical distributed data interface (FDDI) networks.

Characteristic data patterns such as these, include component bit patterns, some of which are common among several characteristic data patterns. For example, a starting delimiter bit pattern is a common component which begins many other characteristic data patterns such as a token, a MAC frame, and a beacon frame in the IEEE 802.5 token ring protocol. The occurrence of multiple component bit patterns in a characteristic data pattern can be generalized by referring to a first component pattern which is followed by a second component pattern.

The adaptive, active monitor comprises two finite state machines (FSM) which are constructed to detect the occurrence of a characteristic data pattern having two consecutive component bit patterns. The first FSM is called the predecessor FSM, and it is configured to detect the first component pattern. The second FSM is called the successor FSM, and it is configured to detect the second component pattern. The first FSM will send a starting signal to the second FSM, when the first FSM has successfully detected the first component pattern. The starting signal initializes the second FSM, to take over the analysis of the portion of the bit stream which follows the first component pattern. If the second FSM successfully detects the second component pattern, it then outputs a pattern alarm signal, indicating the successful detection of the entire characteristic data pattern.

Another feature of the adaptive, active monitor is the accommodation of a component bit pattern which is common to two or more distinctly different characteristic data patterns. For example, a first characteristic data pattern is composed of a first-type component bit pattern followed by a second-type component bit pattern. A second characteristic data pattern is composed of the same first-type component bit pattern followed by a third type component bit pattern. A first FSM is configured to detect the first component pattern, a second FSM is configured to detect the second component pattern, and a third FSM is configured to detect the third component pattern. The objective is to detect either one of the two characteristic data patterns. The predecessor FSM will have a plurality of successor FSMs which run simultaneously in parallel. The first FSM will send a starting signal to both the second FSM and to the third FSM, when the first FSM has successfully detected the first component pattern. The starting signal initializes the second FSM, to take over the analysis of the bit stream which follows the first component pattern, to look for the second component bit pattern. And the starting signal initializes the third FSM, to take over the analysis of the same bit stream which follows the first component pattern, to look for the third component bit pattern. The second FSM and the third FSM run simultaneously in parallel and are mutually independent. They both run until one of them fails or one of them succeeds in finding its designated component bit pattern.

In this manner, the speed of detection of a characteristic data pattern is increased, the number of components is decreased, and effective, real time control can be achieved for high speed data networks.

Still another feature of the adaptive, active monitor is the programmability of the FSMs and the programmability of their interconnection. Each FSM consists of an address register and a memory. The address register has two portions, an n-X bit wide first portion and a X-bit wide second portion X. X is one bit for binary data, X is a word of two bits for Manchester encoded data, or X is a word of five bits for FDDI encoded data. The X-bit wide portion is connected to the input data stream which contains the characteristic data pattern of interest. The n-X bit wide portion contains data which is output from the memory. The next address to be applied by the address register to the memory is made up of the X-1 bit wide portion and the next arriving X-bit word from the input data stream.

Each memory has a plurality of data storage locations, each having a first portion with n-X bits, to be output to the address register as part of the next address. Many of the memory locations have a second portion which stores a command to reset the address register if the FSM fails to detect its designated component bit pattern.

A terminal location in the memory of an FSM will include a start signal value to signal another FSM to

EXHIBIT 13
Part 3 of 4

start analyzing the data stream. If the terminal location in a predecessor FSM memory is successful in matching the last bit of its designated component bit pattern, then it will output a starting signal to a succeeding FSM. The succeeding FSM will begin analyzing the data stream for the next component bit pattern of the characteristic data pattern. The memory of an FSM can be a writable RAM, enabling its reconfiguration to detect different component bit patterns.

Another feature contributing to the programmability of the adaptive, active monitor is the inclusion of a programmable cross point switch, which enables the starting signals from predecessor FSMs to be directed to different successor FSMs. This enables changing the order and combination of FSMs per forming analysis of a bit stream, to detect differently organized characteristic data patterns.

Another feature of the adaptive, active monitor is its functioning in an information collection architecture, to monitor the traffic on a network and to provide event counts for the occurrence of data patterns which are used to control the characteristics of the network.

Further, diverse sized characteristic data patterns can be detected. For example, if when monitoring the 10-bit pattern it is determined that more than 10 bits of information are required, the adaptive monitor feature may be dynamically altered to change the length of the pattern that can be detected. This ability provides increased insight into the characteristics of the data stream.

Another feature of the adaptive, active monitor is its ability to receive serial data streams which include serial patterns of multiple state symbols such as in token ring networks and in fiber optical distributed data interface (FDDI) networks.

The adaptive, active monitor can be embodied as a plurality of FSM integrated circuit chips which are connected in common to receive the input bit stream and which are programmably interconnected to transfer start signals. The adaptive, active monitor can also be embodied as a unitary VLSI circuit chip. And the adaptive, active monitor can also be embodied as a plurality of FSM program task partitions in the memory of a multi-tasking processor.

An additional feature of the adaptive, active monitor is an information collection architecture system, with an adaptable, simultaneously parallel array of finite state machines, for monitoring a data communications network. The system includes an array of at least three finite state machines, embodied on a VLSI circuit chip or alternately in separate task partitions of a multitasked data processor. Each finite state machine in the array, includes a memory, an address register coupled to the network, a start signal input and a pattern detection output coupled to a counter, the memory thereof storing a finite state machine definition for detecting a unique data pattern on the network. Each machine can detect a different pattern. A programmable interconnection means is coupled to the finite state machines in the array, for selectively interconnecting the pattern detection outputs to the start signal inputs. An event vector assembly means, has inputs coupled to the counters, for assembling an event vector from an accumulated count value in the counters, representing a number of occurrences of the data patterns on the network. An information collection means, has an input coupled to the event vector assembly means, an array output coupled to the memories of the finite state machines, and a configuration output coupled to the programmable inter connection means, for receiving the event vector and in response thereto, changing the array to change data patterns to be detected on the network.

The information collection means, in response to receiving the event vector, changes a first interconnection arrangement of the first pattern detection output being connected to the second start signal input, to a second interconnection arrangement of the first pattern detection output being connected to the third start signal input. This changes the composite pattern to be detected. Further, the information collection means, in response to receiving the event vector, changes a first interconnection arrangement of the first pattern detection output being connected to the second start signal input, to a second interconnection arrangement of the first pattern detection output being connected to both the second start signal input and to the third start signal input. This creates a simultaneous, parallel finite state machine operation. Further, the information collection means, in response to receiving the event vector, outputs new finite machine definition data to at least the first memory to change the first data pattern to be detected.

Further, the information collection means is coupled to the network, and in response to receiving the event vector, outputs a control signal to the network to alter communication characteristics thereof. The resulting information collection architecture system provides a flexible, rapidly reconfigurable means to monitor and control data communications networks, through real time monitoring of the data patterns in their traffic.

Now that the uses and features of the adaptive, active monitor have been summarized, the concept and operation of the adaptive, active monitor can be summarized.

The principle of the adaptive active monitoring invention is shown in FIGS. 1A–1F. The adaptive active monitoring invention, monitors a serial data stream having a characteristic pattern. It detects characteristic data patterns in messages on a high speed data network, starting delimiters, tokens, various types of frames such as a MAC frame, a beacon frame, message frames, etc., and other protocol information. Such data streams typically include serial patterns of binary bits. However, some communications protocols, such as the IEEE 802.5 token ring protocol, have multiple state symbols such as the J, K, 0 and 1 symbols (four states), and they can also be accommodated by the invention. The IEEE 802.5 token ring protocol is described in the IEEE Standard 802.5, token ring access method, available from IEEE Incorporated, New York, N.Y., 1989.

Characteristic data patterns such as these include component bit patterns, some of which are common among several characteristic data patterns. For example, a starting delimiter bit pattern is a common component which begins many other characteristic data patterns such as a token, an ending delimiter abort (EDAB) and other communication messages in a token ring protocol. The occurrence of multiple bit patterns in a characteristic data pattern can be generalized by referring to a first component pattern which is immediately followed by a second component pattern for a first characteristic data pattern. A second characteristic data pattern can employ the same first component pattern which will then immediately be followed by a third component pattern which is different from the second component pattern.

In protocols having two characteristic data patterns with some of the component bit patterns being the same, the objective of pattern detection will be to detect either one of the two characteristio datá patterns. In accordance with the adaptive active monitoring invention, the predecessor finite state machine will have a plurality of successor finite state machines which run simultaneously and parallel. The predecessor finite state machine will send a starting signal to both of the successor finite state machines, when the predecessor finite state machine has successfully detected the first component data pattern. The starting signal initializes both of the successor finite state machines to take over the analysis of the bit stream which follows the first component pattern, in order to look for the second component bit pattern or alternately the third component bit pattern. Both successor finite state machines run simultaneously and parallel and are mutually independent. They both run until one of them fails or one of them succeeds in finding its designated component bit pattern. In this manner, the speed of detection of a characteristic data pattern is increased, the number of components of the finite state machine array is decreased, and the effective real time control can be achieved for high speed data networks.

Turning now to FIG. 1A, a parallel finite state machine adaptive monitor 100 is shown. The bit stream 124 which comes from the communications network, is commonly connected to the input of all of the finite state machines FSM 130(J), 130(O), 130(H), 130(N), 130(I), 130(M), 130(E) and 130(B). The starting signal 202 applied to a finite state machine for example FSM 130(J), comes from the termination signal 208 generated by another finite state machine in the array 100. For example, the finite state machine FSM 130(O) has its start signal 202(O) derived from the starting signal 202(0) output by the finite state machine 130(J). The interconnection of the output termination signal 208 of a predecessor finite state machine, to the starting signal input 202 of a successor finite state machine, is accomplished by the programmable cross point switch 210 shown in FIG. 1A. The cross point switch 210 is configured to interconnect the starting signal input of a successor finite state machine to the termination signal output of the predecessor finite state machine, in order to accomplish a desired sequential analysis of component bit patterns making up consecutive portions of a data pattern of interest.

Also included in the adaptive monitor 100 of FIG. 1A, is a global reset signal 204 which is applied to the first occurring finite state machine connected to the bit stream 124. As will be seen in the discussion of FIG. 1E, the first finite state machine will be FSM(F). Also shown in FIG. 1A, is the first FSM 206 designation. Once again, this designation will be applied to one of the plurality of the finite state machines in the array, designating it as the first connected finite state machine to the bit stream being analyzed. The significance of being the first finite state machine is that no starting signal is applied to it. Instead, the first finite state machine connected to a data stream, continuously analyzes the bits in that data stream. All successor finite state machines to the first, predecessor finite state machine, require input starting signals to initiate their respective analyses of the bit stream 124.

FIG. 1A also shows the pattern alarms 144, 144' and 144" which result from the satisfactory completion of the analysis of a corresponding characteristic data pattern. The finite state machines FSM 130(J) through FSM 130(B) in FIG. 1A, can be each embodied as a large scale integrated circuit (LSI) chip, connected by means of a bus for conducting the start signals 202 and the termination signals 208 with the programmable cross point switch 210. The programmable cross point switch 210 can also be a separate LSI circuit chip. In another embodiment of the invention, the finite state machines FSM 130(J) through 130(B) and the programmable cross point switch 210, can collectively be integrated into a very large integrated circuit VLSI circuit chip.

FIG. 1B shows another embodiment of the finite state machine array 100 shown in FIG. 1A, wherein it provides for a large array of selectively interconnectable finite state machines. In FIG. 1B, the programmable cross point switch 210 can selectively interconnect the termination signals from predecessor finite state machines to the start signal inputs 202 of successor finite state machines in a flexible, programmable manner. By configuring the interconnection pattern in the cross point switch 210, predecessor finite state machines may be selected for applying starting signals to successor finite state machines. A particular pattern of interconnection is shown in FIG. 1E and another particular pattern is shown in FIG. 1F. The embodiment shown in FIG. 1B can also be implemented as a plurality of LSI circuit chips or alternately all of the elements shown in FIG. 1B can be integrated onto the same very large scale integrated circuit chip.

FIG. 1C shows a data processor implementation of the adaptive, active monitoring invention. FIG. 1C shows the processor 102 which includes the memory 104 connected by means of the bus 106 to the CPU 108, the disk drive 114, the display and keyboard 116 and the LAN adaptor 118. The LAN adaptor 118 is connected by means of the bit stream 124 to the local area network 140.

The memory 104 includes a bit stream partition 124, and a plurality of task partitions 120F, 120(R), 120(E'), 120(J), 120(O), 120(H), 120(N), 120(I), 120(M), 120(E) and 120(B). Each task partition, such as task partition 120(F) includes a start partition 202, an address register partition 134, and a matrix partition 132. For example, task 120(F) includes the address register 134F' and the matrix 132(F). In order to designate a first FSM, the start partition 122 for the task 120(F) stores the designation "FIRST." The memory 104 also includes a multitasking operation system program 128. In accordance with the invention, each task 120(F), 120(R), 120(E'), 120(J), 120(K), etc., will be executed in parallel in a multi-tasking mode. Each task will have applied to it the current bit from the bit stream 124. The respective task 120 will operate in the same manner as the finite state machines 130(J), 130(O), 130(H), 130(N), 130(I), 130(M), 130(E), 130(B), for example in FIG. 1A. The task 120 in FIG. 1C will pass starting signals from a predecessor finite state machine operating in a first task, for example task (F) to a successor finite state machine operating in a next task, for example task 120(R).

In accordance with the invention, the task 120(J) will issue a starting signal upon satisfactory termination of its matrix 132(J), to three parallel tasks, task 120(O), 120(I), and 120(E). This will launch simultaneous, independent, parallel operation of the tasks 120(O), 120(I) and 120(E) as three parallel successor tasks to the task 120(J).

FIG. 1D illustrates the finite state machine FSM 130(F), which is the first finite state machine connected to the bit stream 124. This can be more clearly seen with reference to FIG. 1E. Referring to FIG. 1E, a standard, modular, finite state machine configuration is shown for the FSM 130(F), the FSM 130(R) and the FSM 130(A). Each modular, finite state machine 130 shown in FIG. 1E, includes a bit stream 124 input, a global reset 204 input, a first FSM 206 input, a start signal 202(F) input, an output data port and an output next start port. By interconnecting a plurality of modular, finite state machines FSM 130(F), 130(R), 130(A), etc. by using the cross point switch 210 in FIG. 1E, a desired interconnection of predecessor and successor finite state machines can be achieved. The register file 138 distributes the global reset input 204 and the first FSM 206 designation. The register 126 stores the identity of the first FSM in the array. For the example given in FIG. 1E, the first FSM is FSM(F). The register file 138 has a selectable output 125 which is controlled by the contents of the register 126, which stores the identity of the first FSM. The designation of first FSM and the global reset input 136 are connected over the output 125 to only one of the finite state machines, based upon the identity of the first FSM in register 126. The rest of the finite state machines in the array are not connected to the global reset input 136 and they do not have the designation as "first FSM." In the example shown in FIG. 1E, the FSM 130(F) is designated as the first FSM and it receives the global reset signal from input 136. The corresponding connections from the register file 138 to the rest of the finite state machines in the array are disabled, in response to the designation in register 126. This prevents a global reset input 204 and a first FSM input 206 from being applied to the FSMs 130(R) and 130(A) in FIG. 1E. Note also that there is no starting signal 202(F) applied to the first FSM 130(F). Starting signals are only applied to successor finite state machines in the array. Thus, the first finite state machine FSM 130(F) continuously analyzes the input bit stream 124, without the necessity of being restarted with a start signal.

It is seen that in FIG. 1E, the output next start signal 202(R) from the first FSM 130(F) is connected by means of the cross point switch 210, to the start signal input of the second FSM 130(R). Similarly, the output next start signal 202(A) from the FSM 130(R) is input to the start signal input 202(A) of the FSM 130(A), by means of the cross point switch 210. The cross point switch 210 is able to selectively reconfigure the interconnection of the finite state machines in the array shown in FIG. 1B. It is seen in FIG. 1E that the register 126 selectively designates one of the finite state machines in the array of FIG. 1B as the first finite state machine. In the example shown in FIG. 1E, the first finite state machine is designated as FSM(F). That first finite state machine will receive the global reset 204 and the first FSM designation 206.

FIG. 1F continues the illustration, showing a significant feature of the invention. The finite state machine FSM 130(J) has an output next start signal which consists of three starting signals, 202(E), 202(I), and 202(O). They respectively start the FSM 130(E), FSM 130(I) and FSM 130(O). This is accomplished by the selective configuration of the cross point switch 210 in FIG. 1F. It is seen that the register 126 continues to designate the FSM(F) of FIG. 1E, as the first FSM. Therefore, no global reset or first FSM designation is given to FSM 130(J), FSM 130(O) or FSM 130(I) of FIG. 1F. Thus, it is the start signal 202(J) which starts the analysis of FSM 130(J) of the bit stream 124. Similarly, it is the start signal 202(O) which starts the analysis by FSM 130(O) of the bit stream 124. Similarly, it is the start signal 202(I) which starts the analysis by FSM 130(I) of the bit stream 124.

Turning now to FIG: 2, it is seen how the finite state machine array is interconnected to perform three parallel data pattern analyses in the bit stream 124. The frame detector 110 consists of the FSM 130(F), 130(R), 130(A), 130(M) and 130(E'). These finite state machines look for the consecutive characters "F", "R", "A", "M" and "E" in the bit stream 124. If they are found, then the start signal 202(J) is passed from the FSM 130(E') to the FSM 130(J).

The frame type detector 112, which consists of the FSM 130(J), 130(O), 130(H), 130(N), 130(I), 130(M), 130(E) and 130(B), performs frame type detection for three frame type character patterns in the bit stream 124, which can immediately follow the "FRAME" frame designation in the bit stream 124. In accordance with the invention, the FSM 130(J) outputs three starting signals, 202(O), 202(I), and 202(E), to start three parallel, simultaneous, independently operating finite state machine sequences shown in FIG. 2. In the example shown in FIG. 2, the frame detector 110 searches for the character pattern "FRAME." If that pattern is found in the bit stream of 124, then the FSM 130(J) starts the three parallel simultaneous sequences which look for the character pattern "JOHN", or "JIM", or "JEB". The output alarm "FRAME_JOHN", is output from the FSM 130(N) if that character pattern is identified. The output alarm "FRAME_JIM", is output from the FSM 130(M), if that character pattern "JIM", is found in the bit stream 124. The output alarm "FRAME_JEB", is output from FSM 130(B), if "JEB" is found in the bit stream 124. Each of these frame type character patterns must follow the first pattern of "FRAME".

FIG. 3 illustrates a security agent (SA) 10 and communicating devices 40 and 41 connected to a bit stream 124. SA 10 monitors bit stream 124 searching for characteristic patterns in data transmitted between communicating devices 40 and 41. In response to a detected characteristic pattern, SA 10 modifies, injects, or deletes information in bit stream 124. For example, SA 10 can produce and transmit a security alert message to one or both of the communicating devices 40 and 41 or to some other device such as a network security manager not shown in FIG. 3. The SA 10 can also send a response message to one of the communicating devices 40 and 41 or to some other device such as a network security manager via a different communication path not shown in FIG. 3. Those skilled in the art will recognize that many possible paths may exist for the SA 10 to transmit responses to other devices or to take some action in response to a detected characteristic pattern by modifying, injecting, or deleting information in bit stream 124.

FIG. 4 illustrates a security agent (SA) 10 connected to a bit stream 124 consisting of an adaptive, active monitor 100 and a responder 300. Adaptive, active monitor 100 detects characteristic data patterns in bit stream 124 which, in turn, causes a pattern alarm signal 144 to be output. Pattern alarm signal 144 activates responder 300, which in turn responds by modifying, injecting, or deleting information in bit stream 124. Further, in accordance with the invention, adaptive, active monitoring means 100 may be capable of detecting a plurality of characteristic patterns, in which case responder 300 is capable of modifying, injecting, and deleting information in bit stream 124 in a plurality of ways, depending on, and in response to said plurality of pattern alarm signals 144, 144', 144" not shown in FIG. 4.

Referring to FIG. 5, one of a plurality of pattern alarm signals (144, 144', ..., 144") is encoded by pattern alarm encoder 301, whereupon the encoded output is stored in one of a plurality of non-volatile registers 303 and a program latch 302 is set. The program latch is used as a means to signal responder 300 to process the received pattern alarm signal. Non-volatile registers 303 contains a set of parameter values that collectively represent those values used to initialize or configure responder 300, such as (1) a security agent identifier which can be used to uniquely identify messages originated by the security agent, (2) an encoded pattern alarm signal identifying the current alarm signal to be processed by responder 300, a (3) sequence number counter that is incremented and transmitted in each message originated by the security agent, and (4) a secret cryptographic key used by the data encryption algorithm 304 to encrypt/decrypt messages. Data encryption algorithm 304 can be any of several cryptographic algorithms such as the Data Encryption Algorithm (DEA) described in American National Standard X3.92-1981, DATA ENCRYPTION ALGORITHM, American National Standards Institute, New York (Dec. 31, 1981). Security alert message transmission means 306 performs the function of transmitting a constructed security alert message over bit stream 124 using a defined protocol and access method. For example, the constructed security alert message can be inserted into bit stream 124, In a token ring, the security alert message transmission means 306 can consist of software and hardware components permitting the transmission of a "unit" of data on the token ring network. On the other hand, security alert message transmission means 306 can transmit a security alert message by intercepting and modifying data within an existing data structure transmitted over bit stream 124. Upon detecting that program latch 302 has been set (e.g., by polling program latch 302 during periods when processor 305 is inactive), processor 305 constructs a security alert message from information stored in non-volatile registers 303 and causes a message authentication code to be calculated on the security alert message by invoking data encryption algorithm 304, passing the so-produced security alert message and message authentication code to security alert message transmission means 306. Security alert message transmission means 306 causes the security alert message and message authentication code to be transmitted via bit stream 124 to a destination device such as a network security manager device.

Processor 305 can consist of a microprocessor, microcode maintained in a read only memory (ROM), and a random access memory (RAM) for storage of intermediate results. In like manner, processor 305 can consist of a collection of parallel and serial finite state machines similar to the Hershey adaptive, active monitor 100. However, microprocessor-ROM-RAM implementation can be most advantageous since unlike the adaptive, active monitor 100, responder 300 is likely to be required to process pattern alarm signals only occasionally in response to detected security events. That is, security events are likely to arrive only after relatively long periods of delay. Thus, an implementation based on parallel FSMs can best be characterized as "overkill" with respect to responder 300.

FIG. 6 is a block diagram illustration of a Responder 300 that processes security events, which are characterized as one of a plurality of possible pattern alarms (144, 144', ..., 144") output by the Adaptive, Active Monitor 100 of FIG. 4. FIG. 6 is a further elaboration of FIG. 5 wherein nonvolatile registers 303 consist of a security agent identifier 321, a security code 322, a sequence number counter 323, and a cryptographic key 325 and processor 305 consists of a security alert message (SAM) production means 331, a message authentication code (MAC) production means 332, and storage for a so-produced security alert message (SAM) 341 and a so-produced message authentication code (MAC) 342.

Referring to FIG. 6, one of a plurality of pattern alarm signals (144, 144', ..., 144") is encoded by pattern alarm encoder 301, whereupon the encoded output is stored in one of a plurality of non-volatile registers 303 and a program latch 302 is set. Upon detecting that program latch 302 has been set (e.g., by polling program latch 302 during periods when processor 305 is inactive), processor 305 performs the following steps. Sequence number counter 325 is incremented and security agent identifier 321, security code 322, and sequence number counter 325 are read from non-volatile registers 303 and processed by security alert message (SAM) production means 331 to produce an output security alert message 341. SAM 341 is next passed to message authentication code (MAC) production means 332 to produce a message authentication code (MAC) 342. In order for Message authentication code (MAC) production means 332 to produce MAC 342, SAM 341 and cryptographic key 327, which is retrieved from non-volatile registers 303, are passed as inputs to data encryption algorithm 304 which performs the individual steps of encryption in order to produce MAC 342. The so-produced SAM 341 and MAC 342 are the provided to security alert message transmission means 306 which causes SAM 341 and MAC 342 to be transmitted via bit stream 124 to a network security manager device.

FIG. 7 depicts a security alert message 341 consisting of a security agent identifier 321, a security code 322 and a sequence number counter 325. Further in accordance with the invention, security code 322 consists of a CATEGORY 323 which provides a broad characterization of the detected pattern alarm and a TYPE 324 which specifies a particular pattern alarm within category 323. For example, the security alert message can define categories such as these (1) virus, (2) inappropriate word usage, (3) intrusion, and (4) non-encrypted text. For CATEGORY = virus, TYPE can be defined as (1) Christmas EXEC, (2) ..., and so forth. Those skilled in the art will recognize (1) there are many ways in which security events can be divided into categories and types and (2) said invention is independent of the categories and types which are selected.

A security agent identifier 321 is included in security alert message 341 so that the messages' receiver will have proof of the identity of the security agent who has detected a pattern alarm and originated the security alert message. A sequence number counter 325 is included in security alert message 341 so that an adversary, who may intercept and replay security alert message 341 in bit stream 124, will be unable to cause the designated receiver to accept the security alert message as genuine. In an alternate embodiment of the invention, a time stamp can be used in place of sequence number counter 325 to prevent message replay attacks.

U.S. Pat. No. 4,918,728 entitled "Data Cryptography Operations Using Control Vectors", cited above under Related Patents and Patent Applications, describes a method for generating message authentication codes using the Data Encryption Algorithm (DEA) and a secret cryptographic key. The method is also described in American National Standard (ANSI) X9.9-1986 Financial Institution Message Authentication (Wholesale). The MAC is a 32-bit cryptographic checksum which cannot be calculated without knowledge of the secret cryptographic key. Thus, cryptographic key 327 must be shared by security agent 10 and the designated receiver (e.g., a network security manager). To verify a MAC, the receiver calculates a MAC-of-reference using his copy of the cryptographic key 327 and his copy of the Data Encryption Algorithm 304 using a comparable message authentication code production means 332 (see FIG. 6). If the received MAC and the calculated MAC-of-reference are equal, the received security alert message is accepted as genuine. Otherwise, if the received MAC and the calculated MAC-of-reference are unequal, the received security alert message is rejected. In effect, the integrity of transmitted security alert messages is assured through the use of the MAC. An adversary cannot change a single bit in the security alert message without causing a completely different, and unpredictable, MAC-of-reference to be generated by the receiver. Without the cryptographic key, it is impossible for an adversary to impersonate an authorized security agent.

In response to received security alert messages, the network security manager can take a number of possible actions. Management can be notified in situations where employees use systems resources for non-business purposes, use inappropriate language in E-mail messages, or violate copyright rules. Management, IS, or site security can be notified in situations where possible intrusion is suspected. A virus alert message can be broadcast to system users if a virus is detected. The network security manager may shut down portions of a system until system files can be scanned for a suspected virus.

FIG. 8 is a block diagram illustration of an alternate embodiment of the invention (as described in FIG. 6) wherein the pattern alarms 144*a*, 144*b*, ..., 144*n* from the Hershey adaptive, active monitor 100 are applied to counters 360*a*, 360*b*, ..., 360*n*, respectively, in order to prevent adaptive, active monitor 100 of FIG. 4 from over-running responder 300. The sizes of counters 360*a*, 360*b*, ..., 360*n* are set so that the number of pattern alarm signals does not cause a counter overflow during the interval of time in which a security alert message is produced and transmitted. For example, 32-bit counters are more than adequate to prevent counters from overflowing.

Referring now to FIG. 8, each pattern alarm signal 144*i* causes its associated counter 144*i* to be incremented by value +1, so that each counter records the numbers of respective pattern alarm signals produced by adaptive, active monitor 100. Processor 305 contains three processing functions, as follows: (1) counter scanning means 333, (2) security alert message production means 331, and (3) message authentication code production means 332. Counter scanning means 333 continually scans the counters, 360*a*, 360*b*, etc., searching for a non-zero counter value. When the final counter 360*n* is reached, the scanning continues with counter 360*a*, and so forth. When a non-zero counter value is detected, the counter value is read out and a security alert message and message authentication code are produced and transmitted. It is assumed that the process of reading out a counter value causes the counter to be reset to zero. In this way, the counter can continue to be updated during the time interval when a security alert message and message authentication code are produced and transmitted. Afterwards, counter scanning means 333 continues searching for a non-zero counter value—starting with the next counter in sequence following the counter that was just processed. Counter scanning means 333 also makes use of an index value representing the index of the counter currently being scanned. Upon detecting a non-zero counter value (via counter scanning means 333), processor 305 performs the following steps:

1. Sequence number counter 325 is incremented.
2. Security agent identifier 321 and sequence number counter 325 are read from nonvolatile registers 303.
3. The value in counter 360*i*, corresponding to index i (where i is the index of the current counter being scanned), is read.
4. Sequence number counter 325, security agent identifier 321, index i, and the value of counter 360*i* are passed to passed as inputs to security alert message production means 331.
5. The value of index i is then used as a means to access a corresponding security code. It is assumed that processor 305 contains a table of n predefined security codes corresponding to the n pattern alarm signals 144*a*, 144*b*, ..., 144*n* and to the n counters 360*a*, 360*b*, ..., 360*n*, respectively. Thus, each index value uniquely specifies a security code 322 consisting of a category 323 and a type 324.
6. Security alert message production means 331 produces a security alert message 341, conforming to the message format shown in FIG. 9, consisting of a security agent identifier 321, a security code 322, a sequence number counter 325, and a pattern alarm counter value 326 corresponding to the value in counter 360*i*.
7. SAM 341 is next passed to message authentication code (MAC) production means 332 to produce a message authentication code (MAC) 342. In order for Message authentication code (MAC) production means 332 to produce MAC 342, SAM 341 and cryptographic key 327, which is retrieved from nonvolatile registers 303, are passed as inputs to data encryption algorithm 304 which performs the individual steps of encryption in order to produce MAC 342.
8. The so-produced SAM 341 and MAC 342 are then provided to security alert message transmission means 306 which causes SAM 341 and MAC 342 to be transmitted via bit stream 124 to a network security manager device.

In yet another alternative embodiment of the invention (not shown in a figure), responder 300 can periodically read out all the counter values and form a single security alert message containing the counter values corresponding to each pattern alarm. In this case, responder 300 would require an internal clock, e.g., a counter that is incremented by 1 for each bit in bit stream 124 that is scanned by adaptive, active monitor 100 of FIG. 4. When the clock reaches a predetermined threshold value, processor 305 would gain control. Processor 305 would then read the counter values, produces a security alert message containing a vector of counter values and a message authentication code, and transmits the security alert message and message authentication code to the network security manager via bit stream 124.

FIG. 9 depicts an extended security alert message 341 consisting of a security agent identifier 321, a security code 322, a sequence number counter 325, and a pattern alarm counter value 326. The extended security alert message 341 of FIG. 9 differs from the security alert message 341 of FIG. 7 in that the extended security alert message 341 of FIG. 9 contains a pattern alarm counter value 326. Pattern alarm counter value 326 represents the number of pattern alarms (with security code 322) detected by security agent 10 of FIG. 4 (with security agent identifier 321). Extended security alert message 341 can also contain a time-stamp instead of a sequence number counter. This would have the added advantage that a network security manager who receives the security alert message can easily calculate a rate (number of occurrences per standard interval of time) at which the security events are occurring.

Those skilled in the art will recognize that the embodiment of FIG. 8 can be easily adapted to detect and respond to security events corresponding to so-called "dirty" words (4-letter words). In that case, a first "dirty" word is associated with pattern alarm 144*a*, a second "dirty" word is associated with pattern alarm 144*b*, and so forth. The security code 322 would consist of a category = "dirty words" and a type indicating the particular "dirty" word to be reported. The pattern alarm counter value 326 would specify the number of such words detected.

Those skilled in the art will also recognize that the embodiment of FIG. 8 can also be configured and adapted to detect use of computing resources for nonauthorized uses such a non-business purposes. This can be done by configuring the adaptive, active monitor to scan for words and phrases that are unlikely to be found in many business environments. For example, the words "Hymn sing", "Be my valentine", "How's the wife and kids" cannot commonly occur in ordinary business correspondence. Of course, in some environments these phrases can be perfectly legitimate. One would tailor the phrases to the particular business environment. One can as well scan for phrases commonly associated with copyright material, such as a warning notice embedded in copyright protected data. This can flag a possible copyright violation. One can also scan for company security designations in transmitted E-mail, such as "Confidential Restricted" which can demand the use of encryption. Therefore, the detection of the phrase "Confidential Restricted" can well signal a possible security violation.

FIG. 10 is a block diagram illustration of another alternate embodiment of the invention wherein the Hershey adaptive, active monitor 100 of FIG. 4 is configured as an intrusion detector and responder 300 is designed to produce and transmit a security alert message whenever the number of detected pattern alarms of a particular type in a given interval of time reaches a prescribed threshold value. Referring to FIG. 10, the Hershey adaptive, active monitor 100 of FIG. 4 is configured to scan bit stream 124 for three characteristic patterns, specified in double quotation marks:

1: "PASSWORD NOT AUTHORIZED"
2. "You entered an invalid login name or password."
3. "Login incorrect."

Each of these characteristic patterns represents a particular system response in an incorrect and invalid login. The phrase "PASSWORD NOT AUTHORIZED" (in Extended Binary Coded Decimal Interchange Code) is an IBM VM message provided to a host-attached workstation if the login sequence fails. The user VM login screen has the following prompts: "USERID ===V" and "PASSWORD ===>," specified in double quotation marks. These standard phrases can also be defined as characteristic patterns, except that in this case the Hershey adaptive, active monitor 100 would track both invalid as well as valid login requests. The phrase "You entered an invalid login name or password." is a standard response in UNIX to a failed login. In this case, the UNIX login screen has the following prompts: "login:" and "Password:". These standard phrases can also be defined as characteristic patterns. In IBM's version of TCP/IP File Transfer Protocol (FTP) for VM, the system prompts the user with the following: "USER (identify yourself to the host):". In the event of a failed login, the phrase "Login incorrect." is displayed to the user. Thus, the characteristic patterns specified in FIG. 10 will detect failed login attempts for (1) VM, (2) UNIX, and TCP/IP FTP for VM. The reader will appreciate that similar characteristic patterns can be specified for other operating systems and systems applications requiring user login. Referring again to FIG. 10, the characteristic data "PASSWORD NOT AUTHORIZED" is associated with pattern alarm 144*a*, the characteristic data "You entered an invalid login name or password." is associated with pattern alarm 144*b*, and the characteristic data "Login incorrect." is associated with pattern alarm 144*c*. In like manner, pattern alarms 144*a*, 144*b*, and 144*c* are uniquely associated with counters 360*a*, 360*b*, and 360*c*, respectively. For example, counter 360*a* contains a value representing the number of pattern alarms 144*a* (corresponding to "PASSWORD NOT AUTHORIZED") received since said counter was last reset to zero. The counters themselves are assumed to be large enough so that they will not overrun. 32-bit counters would be sufficient to prevent such an overrun. In like manner, counter 360*b* contains a value representing the number of pattern alarms 144*b* (corresponding to "You entered an invalid login name or password.") received since said counter was last reset to zero and counter 360*c* contains a value representing the number of pattern alarms 144*c* (corresponding to "Login incorrect.") received since said counter was last reset to zero.

Responder 300 also contains a clock 350 which is attached to Hershey adaptive, active monitor 100 via line 372. Clock 350 is an incrementing counter. For each bit sampled in bit stream 124, a signal is sent via line 372 to clock 350, which causes the clock to increment by +1. When clock 350 reaches a predefined threshold value (e.g., the counter has a high-order one bit), program latch 302 is set. A clock size and threshold value are selected so that the time it takes to produce and transmit a security alert message is less than the time it takes clock 350 to cycle from zero to its threshold value. When processor 305 is not busy producing and transmitting a security alert message, processor 305 is busy monitoring program latch 302. When processor 305 detects that program latch 302 has been set, it reads the counter values (360*a*, 360*b*, and 360*c*), resets the counters to zero, and resets program latch 302. The counters are read and reset before the Hershey adaptive, active monitor is able to send another pattern alarm **144**, thus preventing loss of information. Once the counters (*360a*, *360b*, and *360c*) have been read and reset, Hershey adaptive, active monitor continues, as before, sending pattern alarms (*144a*, *144b*, and *144c*). Processor **305** performs the following steps:

1. Each counter value is compared against a predefined threshold value stored in Threshold **370** (the value in counter *360a* is compared against Ta in Threshold **370**, the value in counter *360b* is compared against Tb in Threshold **370** and the value in counter *360c* is compared against Tc in Threshold **370**).
2. If one or more of the counter values is greater than or equal to its corresponding threshold value (in Threshold **370**), then security alert message production means **331** produces a security alert message **341** of the type shown in FIG. 7, containing (a) a security agent identifier **321**, (b) a security code consisting of a category = "suspicious_login" and a type field signifying which of the counters reached its threshold, and (c) a sequence number counter **325**. Otherwise, if no counter value has reached its threshold, then no security alert message **341** is produced and processor **305** again scans program latch **302** waiting for it to be set.
3. Message authentication code production means **332** now produces a MAC **342** on the so-produced security alert message **341**.
4. Security Alert message transmission means **306** transmits SAM **341** and MAC **341** to the network security manager via bit stream **124**.

FIG. **11** is a block diagram illustration of another alternate embodiment of the invention wherein the Hershey adaptive, active monitor **100** of FIG. 4 is configured to detect the transmission of plain or clear text as opposed to ciphertext. In this security application, it is assumed that encryption is performed at the application layer (or layer 7 of the OSI stack) on the data portions of the transmitted messages. In this case, security agent **10** monitors a communications line to ensure that plaintext is not transmitted in situations where ciphertext is expected to be transmitted. However, there is a complicating factor that must also be addressed, namely that data compression techniques may be employed to compact the transmitted plaintext. Whereas, a relatively simple test can distinguish plaintext from ciphertext, such a test would most likely fail to distinguish compressed plaintext from ciphertext. So there are two cases that must be considered: (1) uncompressed plaintext is transmitted and (2) compressed plaintext is transmitted. We shall consider the first case, which accounts for a large part (and perhaps the largest part) of all text data transmitted in a network. Typically, Electronic Mail (or E-mail), which is a growing networking application, is uncompressed during transmission. Thus, if a link were supposed to carry only encrypted text and through an error or a failure the step was omitted, it would be likely that some uncompressed plaintext would be carried on the bit stream to be monitored-even if compression techniques were employed (e.g., to compress data files for storage and transmission). Thus, the description that follows assumes that compression is not used or is not used universally to compact all data transmitted in the bit stream **124**.

The Hershey, adaptive active monitor **100** is configured to scan for each of the 256 possible 8-bit characters in the data portion of each transmitted frame. The Hershey adaptive active monitor **100** accomplishes this scanning for the starting and ending delimiters for each the data block and then scanning and recording each character within each data block. A method for accomplishing this is taught by Hershey and Waclawsky in copending U.S. patent application entitled "System and Method for Adaptive, Active Monitoring of a Serial Data Stream Having a Characteristic Pattern," Ser. No. 08/138,045 cited above under Related Patents and Patent Applications. Responder **300** is designed to produce and transmit a security alert message whenever the distribution of detected characters in a given interval of time "looks" more like plaintext than ciphertext, which is based on a statistical calculation.

Referring to FIG. **11**, the Hershey, adaptive, active monitor **100** of FIG. **4** is configured to scan bit stream **124** for any of the 256 characters within the data portion of a transmitted frame. Pattern alarm *144a* corresponds to the 1st character, designated B'00000000'; pattern alarm *144b* corresponds to the 2nd character, designated B'00000001', ..., pattern alarm *144n* corresponds to the 256th character, designated B'11111111'. For a given interval of time, counter *360a* records the number of detected characters of the form B'00000000', counter *360b* records the number of detected characters of the form B'00000001', ..., counter *360n* records the number of detected characters of the form B'11111111'.

Responder **300** also contains an accumulator **361**, which is connected to each of the 256 pattern alarm lines (*144a*, *144b*; *144n*). Each pattern alarm causes its associated counter to increment by +1 and it also causes the accumulator **361** to increment by +1. In this manner, accumulator **361** contains a value equal to the sum of the values in the 256 counters (*360a*, *360b*, ...; *360n*). When accumulator **361** reaches a predefined threshold value, program latch **302** is set. An accumulator size and threshold value are selected so that enough characters are sampled from the data portion of the frame or frames to allow a meaningful statistic to be calculated. For example, a few hundred characters would ordinarily suffice to discriminate plaintext from ciphertext, although a value of 1000 shall be selected in order to illustrate the process to be used. When processor **305** is not busy producing and transmitting a security alert message, processor **305** is busy monitoring program latch **302**. When processor **305** detects that program latch **302** has been set, it reads the counter values (*360a*, *360b*, ..., *360n*), resets the counters to zero, and resets program latch **302**. The counters are read and reset before Hershey adaptive, active monitor is able to send another pattern alarm **144**, thus preventing loss of information. Once the counters (*360a*, *360b*, ..., *360n*) have been read and reset, Hershey adaptive, active monitor continues, as before, sending pattern alarms (*144a*, *144b*, ..., *144n*).

Processor **305** first calculates a value phi $=f1(f1-1)+f2(f2-1)+\ldots +f256(f256-1)$, where fi represents the value of the ith counter *360i* read by processor **305**. If the calculated value of phi is greater than $(3902+88t)$, where t is a preestablished value which can be configured within processor **305** then it may be concluded that the sampled text was not random text (or ciphertext). In practice, t would be a value greater than 3 (representing three sigma) and would likely be a value between 5 and 10. In short, t is chosen so as to make it extremely improbable that an observed value of phi $>(3902+88t)$ is the result of sampled random text (or ciphertext). The statistical test described here is based on the phi test described in Solomon Kullback's *Statistical Methods in Cryptanalysis*, pp. 37–39. According to Kullback, the expected value of phi, denoted E(phi), is given by the formula E(phi) =(1/n)(N)(N−1) where n is the number of possible characters and N is the sample size. Substituting n=256 and N=1000, a value of E(phi)=3902 is calculated. Kullback also states that the standard deviation of phi is equal to the square root of (2(n−1)/(n)(n)) * N(N−1). Substituting n=256 and N=1000, the standard deviation of phi is calculated as 88. Roughly speaking about ⅔ of the samples of random text (or ciphertext) will have computed values of phi that lie between 3902−88 and 3902+88, i.e., within one sigma plus or minus of the expected value of phi. The expected value is also called the mean. For practical purposes, nearly all computed values of phi will lie between plus or minus 10 sigma of the mean. Thus to observe a value greater than 10 sigma from the mean would be very unlikely if the sample was taken from random text (or ciphertext). The motivation for the phi test (according to Kullback) is as follows: "It is to be expected, that the variation in the number of occurrences of the n possible elements of a text of N elements would be greater for non-random text than for random text." Essentially, the squaring operation performed in the calculation of phi magnifies the variations in frequency produced by non-random text, thus permitting one to discriminate non-random text from random text.

Consider a plaintext character set consisting of the uppor and lower case alphabetics (52 characters), the numerics (10 characters), and common punctuation (18 characters). In ordinary text produced from this character set there will be a higher concentration of alphabetic characters (mostly lower case), with some numerics and punctuation characters. But suppose for the sake of argument that text produced with this character set is randomly distributed over the character set of 75 characters. Note that E(phi) for random text is less than E(phi) for non-random text. Substituting n=75 and N=1000 in the equations for E(phi) and standard deviation, the value of E(phi)=13320 and the standard deviation of phi is equal to 159. Note that the mean of 13320 for random text over the character set of 75 characters is less than the mean for non-random text over the character set of 75 characters. But the real thing to note here is that 10 standard deviations from E(phi)=3902 for random text over a 256 character set gives a range (3022, 4782) and 10 standard deviations from E(phi)=13320 for random text over a 75 character set gives a range (11730, 14910), which means that these distributions are not overlapping. In this case, there should be no plaintext samples that result in a calculated value of E(phi) that falls in the range (3022, 4782) and there should be no ciphertext sample that result in a calculated value of E(phi) that falls in the range (11730, 14910).

Continuing now with a description of the processing performed by processor 305, if the value of phi is greater than (3902+88t), then security alert message production means 331 produces a security alert message 341 of the type shown in FIG. 7, containing (1) a security agent identifier 321, (2) a security code consisting of a category = "suspicious_plain_text" and a type field set to *NULL*, and (3) a sequence number counter *325*. Otherwise, if the so-calculated value of phi is not greater than (3902+88*t*), then no security alert message is produced and transmitted. In that case, processor 305

again scans program latch 302 waiting for it to be set. Message authentication code production means 332 next produces a MAC 342 on the so-produced security alert message 341. Security Alert message transmission means 306 then transmits SAM 341 and MAC 342 to the network security manager via bit stream 124.

In the situation where ONLY compressed data is transmitted in bit stream 124, it may be possible to adapt the above test to distinguish compressed plaintext from ciphertext. However, this may depend on the quality of the compression being used and it may also depend on a knowledge of the compression algorithm itself in order to construct a testing algorithm that is assured of success. For example, it may be necessary for adaptive, active monitor 100 to be configured to scan for diagrams (2-letter groups) or trigrams (3-letter groups), since the compression algorithm may be very good at "flattening" the frequency distribution of single letters. That is, a test involving only single letters may be insufficient to distinguish compressed *plaintext from* ciphertext, at least for sample sizes which are considered practical. For example, it would do little good if one could distinguish compressed plaintext from ciphertext if it required a sample size of 10 million characters. But by knowing something about the compression algorithm itself, one has a much better chance of constructing a sampling and testing algorithm to detect differences between compressed plaintext and ciphertext. Those skilled in the art will also recognize that from one perspective the goal of compression techniques is to remove the redundancy commonly associated with any ordinary natural language such as English. The more redundancy that one can remove from a compressed text the more it looks like random text or ciphertext. However, an excellent encryption algorithm such as the Data Encryption Algorithm will produce ciphertext that is hardly distinguishable from random text. On the other, a good compression algorithm will produce compressed text that will generally have some, even a small amount, of redundancy laft in it. Hence, the compressed text will not look as much like random text as the ciphertext produced with the DEA will look like random text. This small difference between compressed text and ciphertext is enough to be distinguished provided a large enough sample size is used. In theory, one can distinguish compressed text from ciphertext if a large enough sample size is used. In practice, one can distinguish compressed text from ciphertext only if the compression algorithm is a poor one or some feature of the compression algorithm can be exploited to allow the testing algorithm to operate with only small or modest sample sizes. In summary, the testing procedure described above can in some situations (depending on the compression algorithm) be used to distinguish compressed plaintext from ciphertext—provided a large enough sample size is used. In *other cases*, the *adaptive* action monitor 100 must be reconfigured to scan for particular patterns (such a diagrams and trigrams).

FIG. 12 is an example embodiment of pattern alarms and counters of FIG. 8 configured for virus detection. Referring to FIG. 12, the pattern alarms **144***a*, **144***b*, ..., **144***n* originating with adaptive, active monitor 100 of FIG. 4 are each uniquely associated with a particular characteristic viral pattern. Adaptive, active monitor *100 scans bit stream* **124** *for these virus patterns, which* are character strings or patterns that uniquely identify each virus. As in FIG. 8, the pattern alarms **144***a*, **144***b*, ..., **144***n* from the Hershey adaptive, active monitor 100 are applied to counters 360*a*, 360*b*, ..., 360*n*, respectively. In this way, a clever adversary can not flood the network with viral agents hoping that some will escape detection by over running responder 300. The sizes of counters 360*a*, 360*b*, 360*n* are set so that the number of pattern alarm signals does not cause a counter overflow during the interval of time in which a security alert message is produced and transmitted. For example, 32-bit counters are more than adequate to prevent counters from overflowing. Otherwise, the processing steps to handle a detected virus are the same as those already described in and for FIG. 8.

Those skilled in the art will recognize that the viral patterns described in FIG. 12 may be static patterns or dynamically configurable patterns, depending on how adaptive, active monitor 100 is designed. The Hershey adaptive, active monitor is capable of dynamic re-configuration, thus enabling the security agent 10 to be updated in real time to detect the presence of a suspected offending virus. Following such a re-configuration, which will also cause the type field associated with security code 322 to be adjusted so that each pattern alarm is uniquely associated with a corresponding virus type, both the adaptive, active monitor 100 and responder 300 of security agent 10 of FIG. 4 will operate normally as before.

An example embodiment of pattern alarms and counters for FIG. 8 configured for virus detection, is shown in FIG. 12.

A system and method has been described for monitoring and responding to security events for various network security applications. One mode of response is for the SA to transmit status or commands to other network components via non-network channels. Another mode of response is for the SA to communicate to other network components via the network itself. The following paragraphs explain how this latter mode of communication can be favorably implemented.

FIG. 13 illustrates a general network consisting of a plurality network-attached devices 40, 41, and 42. Devices 40, 41, and 42 are any devices such as hosts, servers, or workstations which communicate with one another via the network 508. Network 508 is a serial bit stream. Network 508 can also be a parallel data stream (e.g., a processor bus). A single SA 10 is attached to the network; SA 10 is configured to monitor the attached network for the occurrence of security events and to respond to the occurrence of security events by transmitting a security alert message to a Network Security Manager NSM 15. The Network Security Manager 15 is a network-attached device (e.g., host or workstation) whose role is to collect security alert messages from one or more SAs in one or more attached networks, and to take action in response to the receipt of the security alert messages. The NSM can be a workstation which displays the occurrence of security events in the network, and permits a Network Security Administrator to issue commands or messages to other network devices in response to received security alert messages.

In order for two network-attached devices such as the SA 10 and NSM 15 to communicate, it is necessary that there exist a network access protocol. A network access protocol, or just protocol, is a set of rules and formats defining the manner in which two network participants can communicate with one another.

FIG. 14 illustrates a simple network architecture consisting of a first application APPL 502 in a first system A 503 communicating with a second application

APPL 504 in a second system B 505 via some network 508. Network 508 here is the serial bit stream illustrated as bit stream 124 in FIG. 3. The skilled reader will appreciate that Network 508 can be a parallel bit stream by making use of the parallel monitoring capabilities of the adaptive, active monitor described by Hershey, et al. in Ser. No. 08/138,045, cited above.

In order to communicate with one another, each system implements the same network access protocol in a network access method. The network access method is typically a combination of hardware and software which is accessible to calling application programs for the purposes of transmitting and receiving data in accordance with a prescribed network access protocol. System A 503 implements the network access protocol in a first network access method NAM 513. System B 505 implements the same network access protocol in a second network access method NAM 515. Although the actual implementations of the network access methods may differ on the two systems, the syntax, semantics, and timing of the implemented protocols must be the same.

In a typical computer communications architecture, the protocols are layered into a structured set of sub-protocols. Each layer performs a specific function of the overall communication task. The Open Systems Interconnection (OSI) model, shown in FIG. 15, depicts one such layered protocol structure.

Note that the number of layers and the functionality assigned to each layer varies depending on the protocol. Transmission Control Protocol/Internet Protocol (TCP/IP) has only four layers. The functions of the OSI application layer, presentation layer, session layer, and transport layer are all handled to some degree by the TCP/UDP layer, layer 4. TCP is the layer 4 protocol for reliable, connection-oriented data transfer. UDP, the User Datagram Protocol, is the layer 4 protocol for connectionless data transfer. FIG. 16 depicts the TCP/IP protocol stack. Most of the variation in modern net working protocols is confined to the upper layers. IEEE 802 standards for layer 2 define the means by which most network access methods implement the low-level media access for token-ring, Carrier-Sense Multiple Access with collision detection (CSMA/CD), Fiber Distributed Data Interface (FDDI), and other network types. Likewise, standards exist which further define the layer 1 signalling techniques and electrical interfaces for each of the network media types (e.g., Manchester encoding).

Data which is transmitted from one system to another progresses down the protocol "stack." At each layer, the output from the previous, upper layer is encapsulated within a larger data block known as a protocol data unit for that layer. For example, the input to layer 3 is a Transport Protocol Data Unit, the output from layer 3 is a Network Protocol Data Unit. The protocol data unit contains the protocol data unit of the previous layer and some header and control information. At the recipient, each layer of the stack removes and interprets the control information, passing the underlying protocol data unit to the next higher layer.

The network access method is typically implemented in a combination of hardware and software. The lower layers of the network protocol stack are typically implemented in hardware on a network interface adapter. The higher layers (e.g., 4 through 7 of the OSI stack) are typically implemented in software in a host processor to which the network adapter is attached. An application programming interface (API) between an application program APPL and layer 7 provides access to various functions of the network access method. The functions may include the ability to establish a session with a remote application, to send or receive messages from a remote application, and to close the session with the remote application.

Furthermore, the communicating applications APPL implement an application communications protocol which defines the syntax, semantics, and timing of messages transmitted from one application to another. For example, the applications may implement a protocol which requires that an application wait to receive a special "polling" request message before transmitting a message to the other application. In a token-ring network, the polling mechanism may be implemented as follows. A first application at a first device periodically broadcasts a poll message. The poll message is a short message containing a flag field and an identifier field. A second application at a second device receives the poll message, removes it from the network, and checks to see if it has any messages to send to the first device. If so, the second application checks to see if the flag field is already set to indicate that some other application at another device has already requested to transmit a message (in which case, the second application must wait for another poll message). If the poll message flag field is not set, the second application modifies the flag field and identifier fields to indicate that the second application has a message it would like to send to the first application. The second application then retransmits the modified poll message to the next device on the ring. The poll message circulates around the ring in this manner until it arrives back at the originating device. The first application receives the modified poll message, removes it from the network, and tests the flag field to determine if any of the devices on the network want to transmit a message. If so, the first application then builds an "envelope" message and transmits it directly to the second application based on the network address associated with the contents of the identifier field of the modified poll message. Upon receipt of the envelope message, the second application removes it from the network, inserts the message it wishes to transmit into a reserved field of the envelope message, and retransmits the envelope message directly to the first application based on the source address contained in the original envelope message. Upon receipt of the envelope message, the first application processes the message contained therein, and broadcasts a new poll message on the network.

A different application communications protocol can be implemented that allows a first application to transmit a new, unsolicited message to a second application. The first application need not wait for an acknowledgment, or a poll message, before transmitting a message to an intended recipient application. Furthermore, the first application can build a new message rather than intercepting and modifying an existing message on the network. The application addresses the message to a second application based on a hard-coded, non-volatile network address. Alternately, the address may be configured at application initialization or installation-time and stored in a volatile or nonvolatile storage.

Yet another application communications protocol can be implemented that allows a first application to multi-cast a message to two or more recipient applications. Again, the distribution list of network addresses is hard-coded in non-volatile storage or it may be configured at application initialization or installation-time and stored in volatile storage. An application multi-casts a single message by transmitting the same message to two or more recipient applications, modifying only the destination address fields of the message.

The interested reader can refer to the text, "Data and Computer Communications," by William Stallings, published by McMillen Publishing Company, New York, N.Y., for additional discussions of network access method implementation and general communications architecture topics.

In the present invention, the application communications protocol is implemented by a Security Alert Message Transfer Method which implements a protocol which permits the SA to transmit security alert messages in an unpolled manner. The skilled reader will appreciate that the Security Alert Message Transfer Method can favorably implement a polled application communications protocol instead.

FIG. 17 is a block diagram showing the Security Alert Message Transmission Means 306 consisting of a Security Alert Message (SAM) Transfer Method 520 and a Network Access Method (NAM) 513. The SAM Transfer Method 520 implements the high-level Security. As described before, the NAM 513 implements the network protocols necessary to route messages in the network.

The network access method to be described for the present invention can take on many forms depending on the network protocol to be implemented. In fact, the embodiment of this invention need only implement those layers of the protocol necessary to route the security alert messages from the SA to the intended recipient (e.g., NSM). For the sake of completeness, two variations of the network access method shall be described: (1) a method that permits routing of security alert messages within a single physical network segment independent of the upper layer protocols in use on that network segment and (2) a method that permits routing of security alert messages through an internetwork of TCP/IP-based devices. In both cases, the communications architecture model to be considered is that of TCP/IP, as illustrated in FIG. 16.

In method 1, all that is required of the network access method is to satisfy the protocol requirements necessary to transmit a message on a single network segment. The network access method of the SA need not concern itself with inter-network routing, session establishment, or other protocol functions above the Data Link Layer (layer 2). The network access method must construct a protocol data unit which satisfies the format requirements of the layer 2 protocol being used in the network. For example, suppose that Network 508 of FIG. 13 is a CSMA/CD Local Area Network (LAN) that complies with IEEE 802.2 for the Logical Link Control (LLC) sub-layer and IEEE 802.3 for the Media Access Control (MAC) sub-layer of the Data Link Layer (layer 2) of the network communications architecture. Note that the MAC referred to here is not to be confused with the Message Authentication Code (MAC) referred to in FIG. 6. These sub-layers are depicted in FIG. 18.

The SAM Transfer Method of the SAM Transmission Means accepts a security alert message and builds a LLC protocol data unit L-PDU. The network access method then encapsulates the L-PDU in a MAC frame. The MAC frame is then transmitted according to the Physical Layer signaling protocol.

FIG. 19 is a block diagram of the LLC sub-layer implementation within the Network Access Method 513 showing the construction of an L-PDU 570 from the input security alert message SAM 341 and the contents of the registers SSAP-REG 561 and DSAP-REG 562 of the Network Access Method 513. SSAP-REG 561 contains the source service access point address. DSAP-REG 562 contains the destination service access point address. Both the SSAP-REG and DSAP-REG register contents are loaded at system configuration time or are hard-coded. The SAM 513 is also input to the Control Field Production Means 563; the output of the Control Field Production Means 563 is put into the CONT field 573 of the L-PDU 570.

FIG. 20 is a block diagram of the MAC sub-layer implementation within the Network Access Method 513 showing the construction of a CSMA/CD MAC frame 590 from the L-PDU 570 constructed above. The rest of the fields of the MAC frame 590 consist of the contents of the pre-configured registers DA-REG 583 and SA-REG 584, the constant preamble PRE 591 and constant starting delimiter SD 592 produced by the Fixed MAC Field Production Means 581, the length field LEN 595 calculated on the input L-PDU 570 by the LEN CALC MEANS 585, the optional padding PAD 597 whose length depends on the contents of the calculated length field LEN 595, and finally the Frame Check Sequence FCS 598 calculated from the input L-PDU 570 by the FCS CALC MEANS 588.

The Network Access Method 513 (see FIG. 17) in a LAN must also follow the media access rules of the MAC sub-layer of the Data Link Layer. In IEEE 802.5 compliant token-ring networks, a station on the ring cannot originate a MAC frame until it has received a "free" token. The token is a special MAC frame which circulates around the ring and is used to control access to the ring. A station which wishes to transmit a message must modify the Access Control field of the token, thus marking it as a "busy" token. The station then retransmits the busy token followed by its message, formatted as a MAC frame as described above and illustrated in FIG. 19 and FIG. 20. Once the busy token and message frame have circulated around the ring (and hopefully been copied by the recipient station addressed in the MAC frame) and have arrived back at the originating station, the originating station can remove the busy token and message frame from the ring and insert a new free token.

In 802.3 compliant CSMA/CD networks, a station on the LAN must listen before transmitting. If the LAN is "quiet," the station may transmit its MAC frame while continuing to listen for "collisions" caused by propagational delays. If a collision is detected, the station jams the network for a certain period then backs off the network for a random period. Once the back-off period is completed, the station is free to contend for the network as before. The interested reader can refer to "Local Networks," by William Stallings for more information on LAN access mediation, token-ring implementation, and other protocol topics.

As described in the book, "Local Networks," by William Stallings, supra the network access method need only implement the bottom two layers (Data Link Layer and Physical Layer) in order to allow messages to be transmitted in a LAN environment. However, the NAM 513 in the present invention may have to implement higher layers in order to interoperate with network devices which implement these higher layer protocols. For example, if the Network 508 to which the SA is attached is interconnected with a second Network 508' via a Layer 3 routing device (e.g., a router), then the NAM 513 must also implement the Network Layer protocols. The protocols at the Network Layer (Layer 3) are used to convey network addressing and control information needed by internetworking devices to route the data from one network, say Network 508, to a second network, say Network 508'. FIG. 21 is a block diagram illustrating an internetwork configuration consisting of an SA 10 attached to a first network 508 communicating by a router 550 to an NSM 15 attached to a second network 508'. The router performs the function of forwarding security alert messages from the SA 10 to the NSM 15. The router also forwards other message types from other network-attached devices 40 in network 508 to one or more network-attached devices 41 and 42 in network 508'. The router allows devices in a network employing one set of network addresses to communicate with devices in a second network employing a second set of network addresses. The router forwards messages based on the source and destination addresses contained in the Network Layer PDU (N-PDU). Thus, an SA 10 wishing to transmit a security alert message through a router to a second network 508' must implement the Network Layer protocol in the network access method. The functions of the network access method 513 are identical to that described for FIG. 19 and FIG. 20 except the security alert message is first encapsulated in a network layer protocol data unit N-PDU not shown. The source and destination network addresses contained in the N-PDU are hard-coded in the network access method 513. The source and destination network addresses may also be retrieved using other network protocols such as ARP (Address Resolution Protocol), as described in "Internetworking with TCP/IP," Volume I, by Douglas E. Comer.

The format of the N-PDU is shown in FIG. 22. In TCP/IP, the N-PDU 600 is also referred to as an Internet Protocol "packet." The fields VERS 601, HLEN 602, SERVICE_TYPE 603, FLAGS 606, TTL 608, PROTOCOL 609, SOURCE_IP_ADDRESS 611, and DESTINATION_IP_ADDRESS 612 are all constructed from pre-configured values or hard-coded constants. The fields TOTAL_LENGTH 604 and HEADER_CHECKSUM 610 may be calculated using means that are not shown but obvious to one skilled in the topic. Likewise, the fields IDENTIFICATION 605 and FRAGMENT_OFFSET 607 may be calculated based on counters and memory registers not shown. Fields IP_OPTIONS 613 and PADDING 614 are typically not required. Guidance on the specific definition and implementation of the IP packet format are further explained by Douglas E. Comer in "Internetworking" with TCP/IP, Volume I.

The DATA 615 field of FIG. 22 consists of the next higher layer protocol data unit. If the SAM Transfer Method is the next higher layer, then the Security Alert Message is inserted in the field DATA 615. In TCP/IP, however, the next higher layer protocol unit is the TCP "segment," hereafter referred to as the Transport PDU or T-PDU. The format of the TCP T-PDU is shown in FIG. 23. The fields SOURCE_PORT 621, DESTINATION_PORT 622, HLEN 625, CODE_BITS 626, and URGENT_POINTER 630 are all constructed from pre-configured values or hard-coded constants. The DATA 633 field contains the Security Alert Message to be transmitted. The fields TCP_OPTIONS 631 and PADDING 632 are typically not needed. All other fields in the T-PDU 620 are constructed using counters, stored values, and simple algorithmic means which are described by Douglas E. Comer in "Internetworking with TCP/IP," Volume I and well-known to those skilled in the topic.

The skilled reader will understand that corresponding functions exist in the Network Access Method of the receiving station (e.g., the NSM) to de-encapsulate first the L-PDU from the received MAC frame, then to de-encapsulate the N-PDU from the L-PDU, then to de-encapsulate the T-PDU from the N-PDU, and finally to de-encapsulate the Security Alert Message from the T-PDU. The symmetric processes of encapsulation at the SA and de-encapsulation at the NSM are illustrated in FIG. 24.

An alternate embodiment of the SAM Transmission Means 306 could be realized by "piggybacking" a simple alert mechanism on top of an existing network protocol which itself may or may not be implemented within the SA 10. In this embodiment, the Security Alert Message takes the form of one or more flag bits which are inserted into a reserved area of an existing network frame. The SAM Transmission Means 306 employs a Hershey adaptive, active monitor to detect a starting delimiter for a free reserved area in an network frame as it passes through the SA 10 network interface. Once detected, if the SA 10 has a security event to report, the adaptive, active monitor can trigger a parallel FSM which modifies one or more bits in the frame prior to its retransmission at the network interface. Other network-attached devices employ similar devices to detect the presence of the predefined flag bit settings and take action in response. The device may in turn activate a program in an attached processor to construct a second, detailed Security Alert Message and transmit the message to other network devices or the Network Security Manager. This embodiment has the advantages that the SAM Transmission Means need not implement a protocol stack, it makes more efficient use of network bandwidth, and it provides a simple mechanism to distribute the responding means to multiple receiving devices in cases where the SA 10 lacks the processing resources to act as a full-fledged network participant. Furthermore, the use of the Hershey adaptive, active monitor to detect the reserved field starting delimiter and to modify the flag bits of the reserved field allow the use of this technique in a high speed communications environment.

Although specific embodiments of the invention have been disclosed, it will be understood by those having skill in the art, that changes can be made to those embodiments, without departing from the spirit and the scope of the invention.

What is claimed is:

1. A network security architecture system, with an adaptable, simultaneously parallel array of finite state machines, for monitoring the security of a data communications network, comprising:

a first finite state machine in said array, including a first memory, a first address register coupled to said network, a first start signal input and a first security threat pattern detection output coupled to a first counter, said memory thereof storing a first finite state machine definition for detecting a first data security threat pattern on said network;

a second finite state machine in said array, including a second memory, a second address register coupled to said network, a second start signal input and a second security threat pattern detection output coupled to a second counter, said memory thereof storing a second finite state machine definition for detecting a second data security threat pattern on said network;

a third finite state machine in said array, including a third memory, a third address register coupled to said network, a third start signal input and a third security threat pattern detection output coupled to a third counter, said memory thereof storing a third finite state machine definition for detecting a third data security threat pattern on said network;

a programmable interconnection means coupled to said first, second and third finite state machines, for selectively interconnecting said first security threat pattern detection output to at least one of said second and third start signal inputs;

a security event vector assembly means, having inputs coupled to said first, second and third counters, for assembling a security event vector from an accumulated count value in said first counter and at least one of said second and third counters, representing a number of occurrences of said first data security threat pattern and at least one of said second and third data security threat patterns on said network; and

a responding means, having an input coupled to said security event vector assembly means, an array output coupled to said memory of said first, second and third finite state machines, and a configuration output coupled to said programmable interconnection means, for receiving said security event vector and in response thereto, changing said array to change data security threat patterns to be detected on said network.

2. The system of claim 1, wherein said responding means, in response to receiving said security event vector, changes a first interconnection arrangement of said first security threat pattern detection output being connected to said second start signal input, to a second interconnection arrangement of said first security threat pattern detection output being connected to said third start signal input.

3. The system of claim 1, wherein said responding means, in response to receiving said security event vector, changes a first interconnection arrangement of said first security threat pattern detection output being connected to said second start signal input, to a second interconnection arrangement of said first security threat pattern detection output being connected to both said second start signal input and to said third start signal input, for simultaneous, parallel finite state machine operation.

4. The system of claim 1, wherein said responding means, in response to receiving said security event vector, outputs new finite machine definition data to at least said first memory to change said first data security threat pattern to be detected.

5. The system of claim 1, wherein said responding means is coupled to said network, and in response to receiving said security event vector, outputs a control signal to said network to alter communication characteristics thereof.

6. The system of claim 1, wherein said security threat is the occurrence of inappropriate words.

7. The system of claim 1, wherein said security threat is the occurrence of a proprietary notice.

8. The system of claim 1, wherein said security threat is the occurrence of use of computing resources for non-authorized purposes.

9. The system of claim 1, wherein said security threat is the occurrence of copyright protected data.

10. The system of claim 1, wherein said security threat is the occurrence of company confidential information.

11. The system of claim 1, wherein said security threat is the occurrence of unencrypted data.

12. The system of claim 1, wherein said security threat is the occurrence of a plurality of intrusions by an adversary who attempts to gain access to a system using repeated login requests.

13. The system of claim 1, wherein said first, second and third finite state machines are formed on an integrated circuit chip.

14. The system of claim 1, wherein said first, second and third finite state machines are formed in task memory partitions of a multi-tasking data processor.

15. The system of claim 1, which further comprises:

said responding means receiving said computer security event vector and in response thereto, performing security monitoring and control operations on said data communications network.

16. The system of claim 15, which further comprises:

said responding means receiving said computer security event vector and in response thereto, transmitting a security alarm signal on said network.

17. The system of claim 1, wherein said security threat is the occurrence of a natural language pattern.

18. The system of claim 1, wherein said security threat is an intrusion detection pattern.

19. The system of claim 1, wherein said security threat is the occurrence of plaintext in said data communications network, where ciphertext is required to maintain security.

20. In a network security architecture system, with an adaptable, simultaneously parallel array of finite state machines, a method for monitoring the security of a data communications network, comprising:

storing a first finite state machine definition for detecting a first data security threat pattern on said network, in a first finite state machine in said array, including a first memory, a first address register coupled to said network, a first start signal input and a first security threat pattern detection output coupled to a first counter;

storing a second finite state machine definition for detecting a second data security threat pattern on said network, in a second finite state machine in said array, including a second memory, a second address register coupled to said network, a second start signal input and a second security threat pattern detection output coupled to a second counter;

storing a third finite state machine definition for detecting a third data security threat pattern on said network, a third finite state machine in said array, including a third memory, a third address register coupled to said network, a third start signal input and a third security threat pattern detection output coupled to a third counter;

selectively interconnecting said first security threat pattern detection output to at least one of said second and third start signal inputs;

assembling a security event vector from an accumulated count value in said first counter and at least one of said second and third counters, representing a number of occurrences of said first data security threat pattern and at least one of said second and third data security threat patterns on aid network; and

receiving said security event vector and in response thereto, changing said array to change data security threat patterns to be detected on said network.

21. The system of claim 20, wherein in response to receiving said security event vector, changing a first interconnection arrangement of said first security threat pattern detection output being connected to said second start signal input, to a second interconnection arrangement of said first security threat pattern detection output being connected to said third start signal input.

22. The method of claim 20, wherein in response to receiving said security event vector, changing a first interconnection arrangement of said first security threat pattern detection output being connected to said second start signal input, to a second interconnection arrangement of said first security threat pattern detection output being connected to both said second start signal input and to said third start signal input, for simultaneous, parallel finite state machine operation.

23. The method of claim 20, wherein in response to receiving said security event vector, outputting new finite machine definition data to at least said first memory to change said first data security threat pattern to be detected.

24. The method of claim 20, wherein in response to receiving said security event vector, outputting a control signal to said network to alter communication characteristics thereof.

25. The method of claim 20, wherein said security threat is the occurrence of inappropriate words.

26. The method of claim 20, wherein said security threat is the occurrence of a proprietary notice.

27. The method of claim 20, wherein said security threat is the occurrence of use of computing resources for non-authorized purposes.

28. The method of claim 20, wherein said security threat is the occurrence of copyright protected data.

29. The method of claim 20, wherein said security threat is the occurrence of company confidential information.

30. The method of claim 20, wherein said security threat is the occurrence of unencrypted data.

31. The method of claim 20, wherein said security threat is the occurrence of a plurality of intrusions by an adversary who attempts to gain access to a system using repeated login requests.

32. The method of claim 20, wherein said first, second and third finite state machines are formed on an integrated circuit chip.

33. The method of claim 20, wherein said first, second and third finite state machines are formed in task memory partitions of a multi-tasking data processor.

34. The method of claim 20, which further comprises:

receiving said computer security event vector and in response thereto, performing security monitoring and control operations on said data communications network.

35. The method of claim 34, which further comprises:

receiving said computer security event vector and in response thereto, transmitting a security alarm signal on said network.

36. A method for information collection by adaptive, active security monitoring of a serial stream of data having a characteristic virus pattern including a first occurring and a second occurring virus pattern portions, comprising the steps of:

receiving x-bit words from said serial data stream, in a first n-bit address register having a first portion with n-x bits and an second portion with two bits and an input to said second portion coupled to said serial stream;

accessing a first-addressable memory having a plurality of data storage locations, each having a first portion with n-x bits, said first memory having an n-bit address input coupled an output of said first address register, said first memory configured with data stored in first and second ones of said data storage locations to represent a first digital filter for said first occurring virus pattern;

transferring data over a feedback path from an output of said first memory to an input of said first register, for transferring said data from said first one of said data storage locations in said first memory to said first portion of said first address register, for concatenation with said x-bit words from said serial bit stream to form an address for said second one of said data storage locations of said first memory;

outputting a start signal from said second one of said data storage locations of said first memory having a start signal value stored therein, which is output when said first occurring portion of said characteristic virus pattern is detected by said digital filter;

receiving x-bit words from said serial data stream, in a second p-bit address register having a first portion with p-x bits and a second portion with a plurality of x bits and an input to said second portion coupled to said serial stream, for receiving x-bit words from said serial data stream;

said second address register coupled to said first memory, for receiving said start signal value from said first memory;

accessing a second addressable memory having a plurality of data storage locations, each having a first portion with p-x bits, said second memory having a p-bit address input coupled an output of said second address register, said second memory configured with data stored in first and second ones of said data storage locations to represent a second digital filter for said second occurring virus pattern;

transferring data over a feedback path from an output of said second memory to an input of said second register, for transferring said data from said first one of said data storage locations in said second memory to said first portion of said second address register in response to said start signal value, for concatenation with said x-bit words from said serial data stream to form an address for said second one of said data storage locations of said second memory;

outputting a security alarm value from said second one of said data storage locations of said second memory having a virus pattern security alarm value stored therein, which is output when said second portion of said characteristic virus pattern is detected by said second digital filter;

counting occurrences of said second portion of said characteristic virus pattern in said data stream, with a counter coupled to said virus pattern security alarm value output, and outputting a count value as an event vector; and

receiving said event vector and in response thereto, performing security monitoring and control operations on a data communications medium providing said data stream.

37. The method of claim 36, wherein said first address register and said first addressable memory are formed in a first multi-tasking memory partition of a data processor; and

said second address register and said second addressable memory are formed in a second multi-tasking memory partition of said data processor, coupled to said first memory partition.

38. The method of claim 36, which further comprises:

reconfiguring said first addressable memory by storing new data in said first and second ones of said data storage locations, to represent a third digital filter for a third occurring virus pattern, in response to said event vector.

39. The method of claim 36, which further comprises:

decoupling said first memory from said second address register and coupling said first memory to a third address register, for receiving said start signal value from said first memory, in response to said event vector.

40. A method for adaptive, active security monitoring of a serial stream of data having a characteristic virus pattern including a first occurring and a second occurring virus pattern portions, comprising the steps of:

receiving x-bit words from said serial data stream, in a first n-bit address register having a first portion with n-x bits and an second portion with two bits and an input to said second portion coupled to said serial stream;

accessing a first addressable memory having a plurality of data storage locations, each having a first portion with n-x bits, said first memory having an n-bit address input coupled an output of said first address register, said first memory configured with data stored in first and second ones of said data storage locations to represent a first digital filter for said first occurring virus pattern;

transferring data over a feedback path from an output of said first memory to an input of said first register, for transferring said data from said first one of said data storage locations in said first memory to said first portion of said first address register, for concatenation with said x-bit words from said serial bit stream to form an address for said second one of said data storage locations of said first memory;

outputting a start signal from said second one of said data storage locations of said first memory having a start signal value stored therein, which is output when said first occurring portion of said characteristic virus pattern is detected by said digital filter;

receiving x-bit words from said serial data stream, in a second p-bit address register having a first portion with p-x bits and a second portion with a plurality of x bits and an input to said second portion coupled to said serial stream, for receiving x-bit words from said serial data stream;

said second address register coupled to said first memory, for receiving said start signal value from said first memory;

accessing a second addressable memory having a plurality of data storage locations, each having a first portion with p-x bits, said second memory having a p-bit address input coupled an output of said second address register, said second memory configured with data stored in first and second ones of said data storage locations to represent a second digital filter for said second occurring virus pattern;

transferring data over a feedback path from an output of said second memory to an input of said second register, for transferring said data from said first one of said data storage locations in said second memory to said first portion of said second address register in response to said start signal value, for concatenation with said x-bit words from said serial data stream to form an address for said second one of said data storage locations of said second memory;

outputting an security alarm value from said second one of said data storage locations of said second memory having a virus pattern security alarm value stored therein, which is output when said second portion of said characteristic virus pattern is detected by said second digital filter.

41. The method of claim 40, wherein said first address register and said first addressable memory are formed in a first multi-tasking memory partition of a data processor; and

said second address register and said second addressable memory are formed in a second multi-tasking memory partition of said data processor, coupled to said first memory partition.

42. An information collection architecture system for adaptive, active security monitoring of a serial stream of data having a characteristic virus pattern including a first occurring and a second occurring virus pattern portions, for performing security monitoring and control operations on a data communications medium providing said data stream, comprising:

a first n-bit address register having a first portion with n-x bits and an second portion with a plurality of x bits and an input to said second portion coupled to said serial stream, for receiving x-bit words from said serial data stream;

first addressable memory having a plurality of data storage locations, each having a first portion with n-x bits, said first memory having an n-bit address input coupled an output of said first address register, said first memory configured with data stored in first and second ones of said data storage locations to represent a first digital filter for said first occurring virus pattern;

a feedback path from an output of said first memory to an input of said first register, for transferring said data from said first one of said data storage locations in said first memory to said first portion of said first address register, for concatenation with said x-bit words from said serial data stream to form an address for said second one of said data storage locations of said first memory;

said second one of said data storage locations of said first memory having a start signal value stored therein, which is output when said first occurring portion of said characteristic virus pattern is detected by said digital filter;

a second p-bit address register having a first portion with p-x bits and a second portion with a plurality of x bits and an input to said second portion coupled to said serial stream, for receiving x-bit words from said serial data stream;

said second address register coupled to said first memory, for receiving said start signal value from said first memory;

second addressable memory having a plurality of data storage locations, each having a first portion with p-x bits, said second memory having a p-bit address input coupled an output of said second address register, said second memory configured with data stored in first and second ones of said data storage locations to represent a second digital filter for said second occurring virus pattern;

a feedback path from an output of said second memory to an input of said second register, for transferring said data from said first one of said data storage locations in said second memory to said first portion of said second address register in response to said start signal value, for concatenation with said bit from said serial data stream to form an address for said second one of said data storage locations of said second memory;

said second one of said data storage locations of said second memory having a virus pattern security alarm value stored therein, which is output when said second portion of said characteristic virus pattern is detected by said second digital filter;

a counter coupled to said virus pattern security alarm value output, for counting occurrences of said second portion of said characteristic virus pattern in said data stream, and outputting a count value as an event counter; and

control means, coupled to said counter, for receiving said event vector and in response thereto, performing security monitoring and control operations on a data communications medium providing said data stream.

43. The system of claim 42, wherein said first address register and said first addressable memory are formed in a first integrated circuit chip; and

said second address register and said second addressable memory are formed in a second integrated circuit chip, coupled to said first integrated circuit chip.

44. The system of claim 42, wherein said first address register and said first addressable memory are formed in a first multi-tasking memory partition of a data processor; and

said second address register and said second addressable memory are formed in a second multi-tasking memory partition of said data processor, coupled to said first memory partition.

45. The system of claim 42, wherein said characteristic virus pattern is from a fiber optical distributed data interface (FDDI) data communications medium.

* * * * *

EXHIBIT 14

US006263442B1

(12) **United States Patent**
Mueller et al.

(10) **Patent No.: US 6,263,442 B1**
(45) **Date of Patent: *Jul. 17, 2001**

(54) **SYSTEM AND METHOD FOR SECURING A PROGRAM'S EXECUTION IN A NETWORK ENVIRONMENT**

(75) Inventors: Marianne Mueller, Woodside; David Connelly, Los Altos, both of CA (US)

(73) Assignee: Sun Microsystems, Inc., Mountain View, CA (US)

(*) Notice: This patent issued on a continued prosecution application filed under 37 CFR 1.53(d), and is subject to the twenty year patent term provisions of 35 U S C 154(a)(2)

Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: 08/652,703

(22) Filed: May 30, 1996

(51) **Int. Cl.**^{7} **G06F 11/30**
(52) **U.S. Cl.** 713/201
(58) **Field of Search** 395/187.01, 186, 395/182 02, 183.14, 200.02; 364/280, 230. 280 6, 281 3, 281 7, 230 3, 286.4; 380/4, 25, 23

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,713,753	*	12/1987	Boebert et al.	711/164
4,780,821	*	10/1988	Crossley	395/670
4,849,877	*	7/1989	Bishop et al.	395/200.56
4,891,785	*	1/1990	Donohoo	395/2(X).47
5,164,988	*	11/1992	Matyas et al.	380/25
5,450,567	*	9/1995	Mori et al.	395/500
5,495,533	*	2/1996	Linehan et al.	380/21
5,572,673	*	11/1996	Shurts	395/186
5,577,209	*	11/1996	Boyle et al	713/201
5,682,514	*	10/1997	Yohe et al	395/445
5,689,708	*	11/1997	Regnier et al.	395/682
5,699,518	*	12/1997	Held et al	375/200.11
5,724,425	*	3/1998	Chang et al	380/25
5,864,683	*	1/1999	Boebert et al.	709/249
5,940,591	*	8/1999	Boyle et al.	713/201

OTHER PUBLICATIONS

Chung, K.-M., et al, "A 'Tiny' Pascal Comiler, Part 1: The P-Code Interpreter," Byte Publications Inc. (1978), pp. 58-65, 148-155

Chung, K.-M., et al , "A 'Tiny' Pascal Complier, Part 2: The P-Compiler," Byte Publications Inc (1978), pp. 34-52.

Thompson, K , "Regular Expression Search Algorithm," *Communication of the ACM* (1968), vol 11, No. 6, pp. 419-422

Mitchell, J G , et al , *Mesa Language Manual*. a Xerox Corp. document

McDaniel, G., An Analysis of a Mesa Instruction Set (1982), a Xerox Corp. document

Pier, A.P., A Retrospective on the Dorado, A High-Performance Personal Computer (1983), a Xerox Corp. document.

Pier, A P., A Retrospective on the Dorado, A High-Performance Personal Computer, Conference Proceedings, The 10th Annual International Symposium on Computer Architecture, Computer Society Press (1983), pp. 252-269

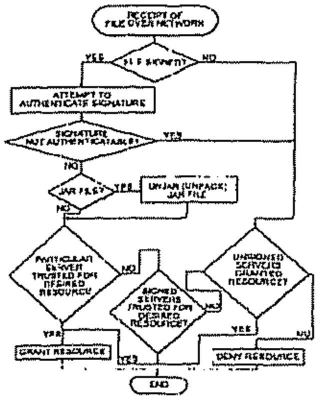
* cited by examiner

Primary Examiner—Dieu-Minh T. Le
(74) *Attorney, Agent, or Firm*—Finnegan, Henderson, Farabow, Garrett & Dunner, L.L.P.

(57) **ABSTRACT**

A system and method for securing a program's execution in a network environment is presented A first server is configured to permit execution of a program from a second server based on a configurable security characteristic of the program The first server receives the program transferred from the second server. Subsequently, the program is checked for the configurable security characteristic The program is executed on the first server if permitted by the configurable security characteristic.

26 Claims, 3 Drawing Sheets


RECEIPT OF FILE OVER NETWORK
FILE SIGNED?
YES
NO
ATTEMPT TO AUTHENTICATE SIGNATURE
SIGNATURE NOT AUTHENTICATABLE?
YES
NO
JAR FILE?
YES
UNJAR (UNPACK) JAR FILE
NO
PARTICULAR SERVER TRUSTED FOR DESIRED RESOURCE
NO
SIGNED SERVERS TRUSTED FOR DESIRED RESOURCE?
NO
UNSIGNED SERVERS GRANTED RESOURCE?
YES
NO
YES
YES
GRANT RESOURCE
DENY RESOURCE
END

Defendant's Trial Ex.
DTX - 1023
Case No. 06-369 GMS

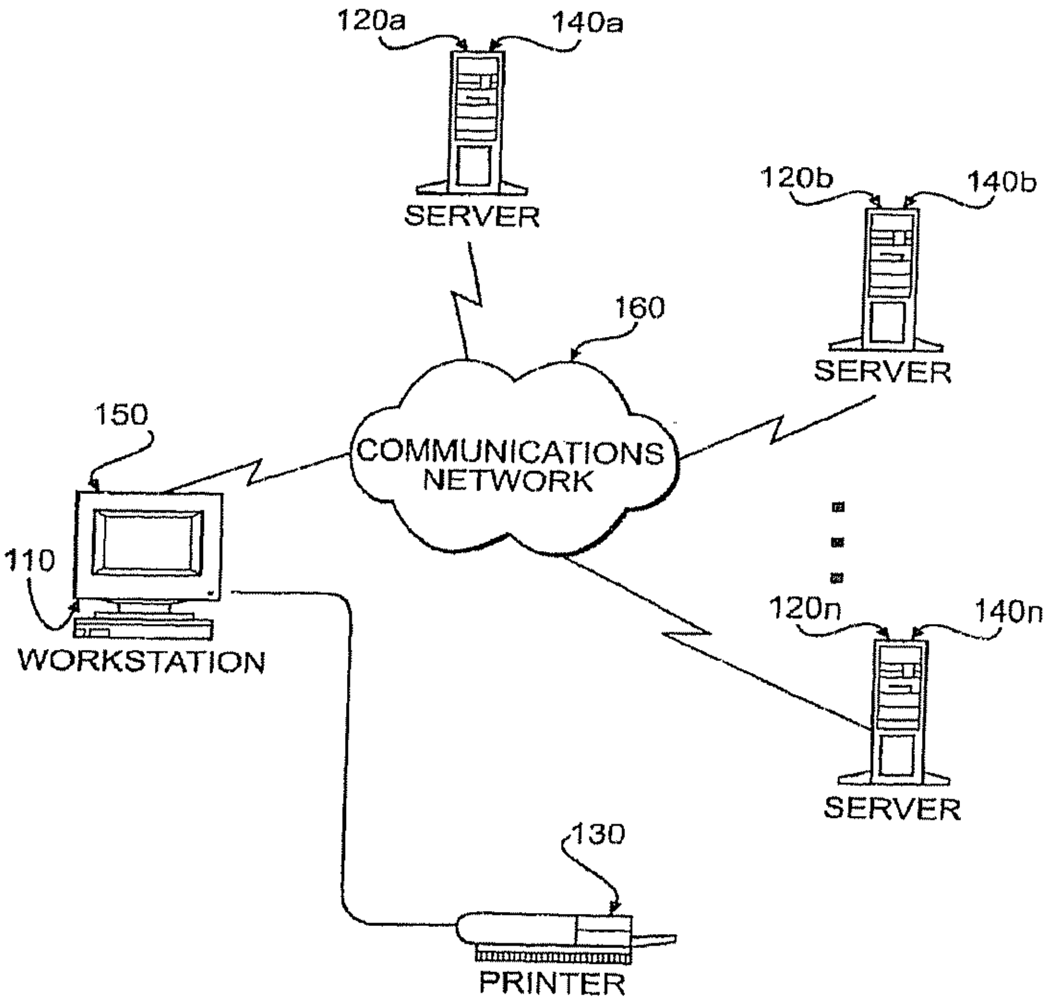


FIG. 1

PRIOR ART

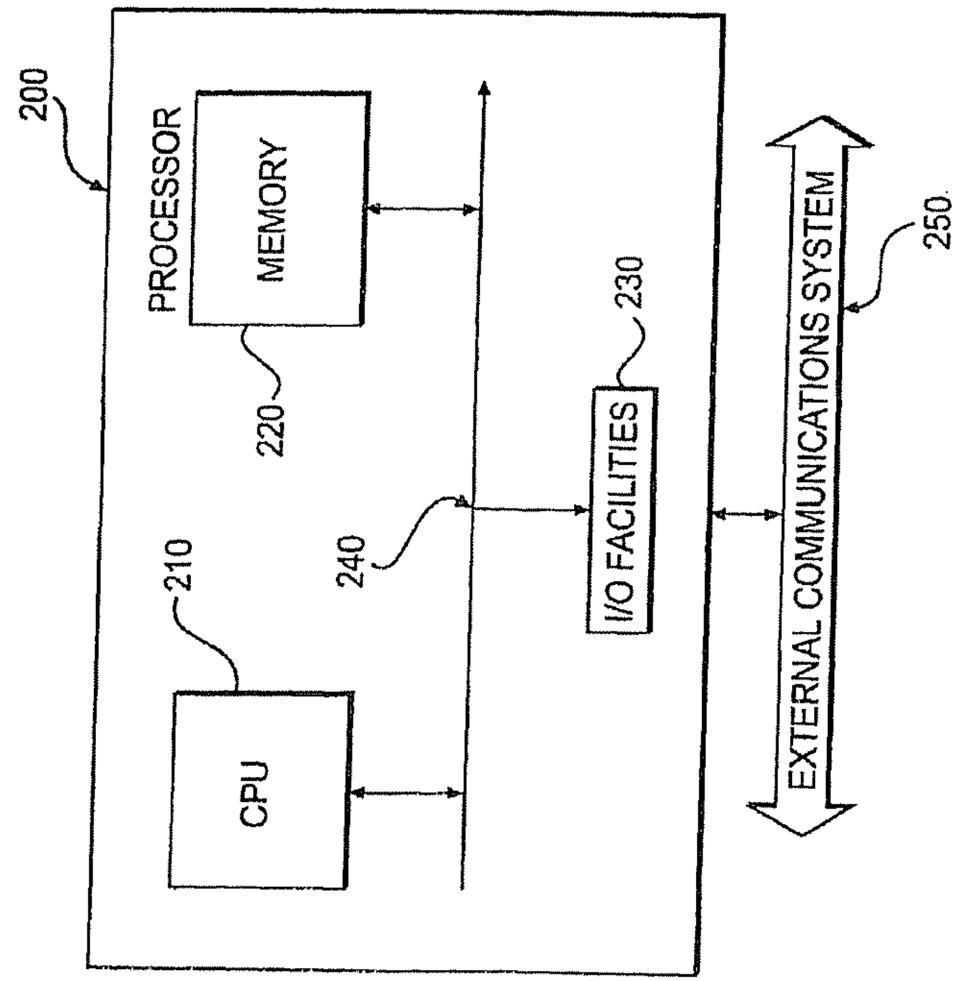

200
PROCESSOR
MEMORY
220
210
CPU
240
230
I/O FACILITIES
EXTERNAL COMMUNICATIONS SYSTEM
250

FIG. 2

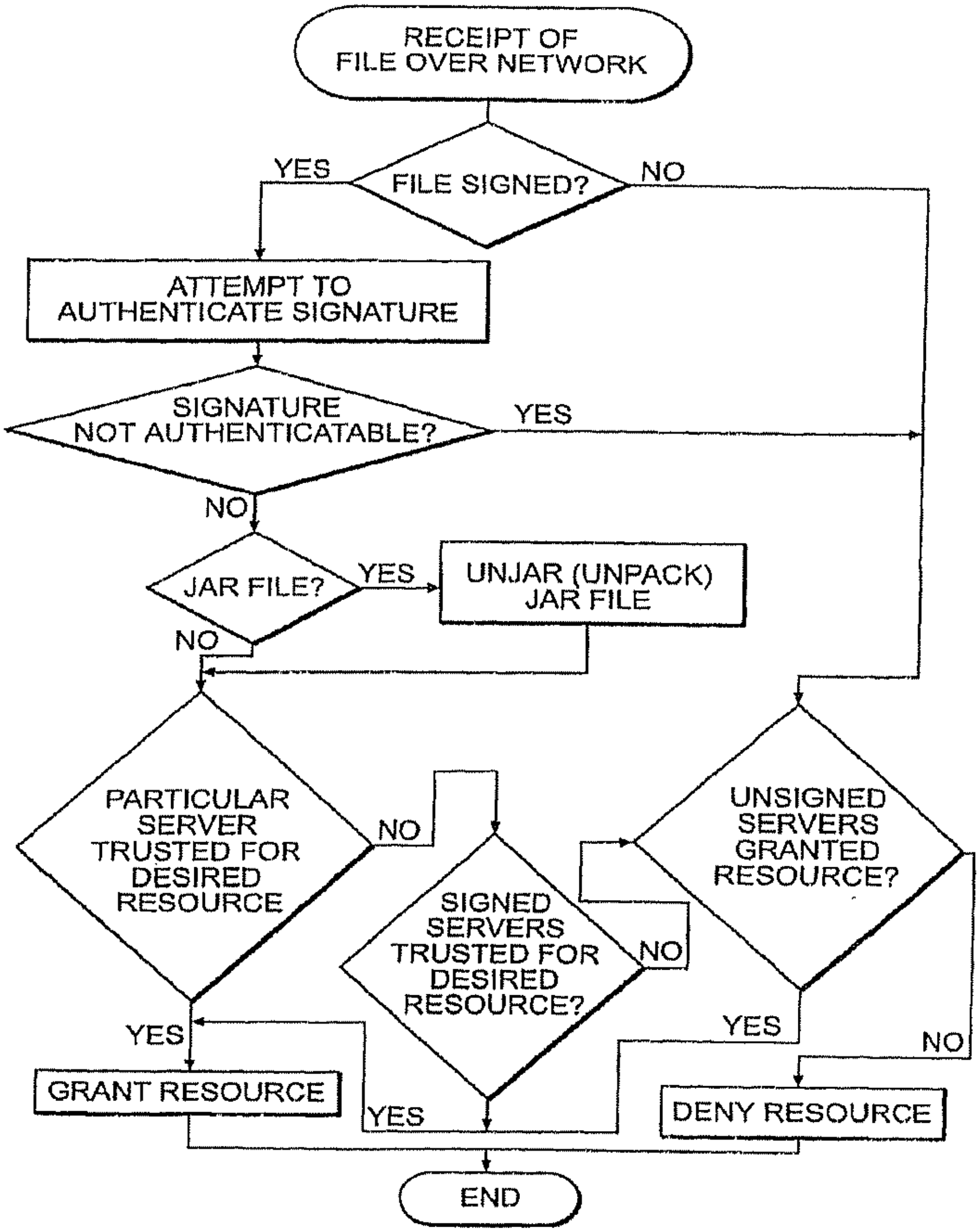

RECEIPT OF FILE OVER NETWORK
FILE SIGNED?
YES
NO
ATTEMPT TO AUTHENTICATE SIGNATURE
SIGNATURE NOT AUTHENTICATABLE?
YES
NO
JAR FILE?
YES
UNJAR (UNPACK) JAR FILE
NO
PARTICULAR SERVER TRUSTED FOR DESIRED RESOURCE
NO
SIGNED SERVERS TRUSTED FOR DESIRED RESOURCE?
NO
UNSIGNED SERVERS GRANTED RESOURCE?
YES
YES
GRANT RESOURCE
YES
NO
DENY RESOURCE
END

FIG. 3

SYSTEM AND METHOD FOR SECURING A PROGRAM'S EXECUTION IN A NETWORK ENVIRONMENT

BACKGROUND OF THE INVENTION

This application relates to the provision of services in a client-server context. More particularly, this application relates to securing inter-server services on behalf of a client over a network.

FIG. 1 illustrates a typical client-server environment within the World Wide Web. As one of ordinary skill in the art will readily appreciate, a user's accessing a web page on the World Wide Web involves the cooperation of (at least) two pieces of software: the web browser **110**, typically directly under the user's control as software on the workstation **150**, and the server **120** for the web page. Responding in a manner predetermined by the author of the web page to transactions initiated by the browser **110**, the server **120** typically resides on a separate processor **140**.

FIG. 2 illustrates a processor **200** such as a workstation **150** or server **120**. Such a processor includes a CPU **210** to which a memory **220** and I/O facilities **230** connect by a bus **240**. The processor **200** connects to an external communications system **250** which is, for example, a network or modem communications link.

As the HyperText Markup Language (HTML) is the preferred language for authoring web pages, the description below is in the terms of HTML. These terms are explained in, for example, I. S. Graham, *The HTML Sourcebook*, 1996 (John Wiley & Sons, Inc., 2d Edition). Graham is incorporated herein by reference to the extent necessary to explain these terms. However, Graham is not prior art.

In addition to text and static images for display on the user's workstation **150** via the user's browser **110**, a web page can also include an applet. An applet is a program included in an HTML page, whose execution a user can observe via a browser **110** enabled to recognize, download and execute the applet and to display the results of the applet's execution. The HotJava™ browser, available from the assignee of the instant invention, is the preferred browser **110**, and the Java™ environment, also available from the assignee of the instant invention, is the preferred environment for encoding and executing applets.

The Java environment is described in, for example, *Java Unleashed* (Sams.net Publishing, 1996). *Java® Unleashed* is incorporated herein by reference to the extent necessary to explain the Java environment. However, *Java® Unleashed* is not prior art. Java and Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

An applet typically is a small program residing on a server **120**. Some HTML document refers to the applet using the <applet> tag. When a browser downloads the HTML document and recognizes the <applet> tag, it also downloads the applet identified by the applet tag and executes that applet.

Written in a general purpose language such as Java, an applet is in this way unrestrained in its functionality. It can perform any function which a program written in any other general purpose language (such as C or PL1) can accomplish. The methodologies of applets, however, are constrained by the Java environment in order to minimize the security risks an applet presents to the workstation **150**. That is to say, an applet is restricted to "play" within a bounded "sandbox."

While a security policy may suffice for the transfer of code from a server to a client, the transfer of code for execution from one server to another server presents greater security risks and requires a more stringent security policy. Accordingly, there is a need for a managing security on a server which receives code for execution.

SUMMARY OF THE INVENTION

Herein is disclosed, in a network environment, a security manager residing on a server and deciding whether to permit the execution of a servlet based on a characteristic of the servlet.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 illustrates a typical client-server environment within the World Wide Web.

FIG. 2 illustrates a processor such as a workstation or server.

FIG. 3 is a flowchart illustrating the flow of command according to an embodiment of the invention.

DESCRIPTION OF THE PREFERRED EMBODIMENT

U.S. patent application Ser. No. 08/657,712 (abandoned) (filed May 30, 1996, entitled, "Method and System for Facilitating Servlets," naming as inventors Pavini P. Diwanji et al., assigned as well to the assignee of the instant patent application) describes a servlet. U.S. patent application Ser. No. 08/657,712 is incorporated herein by reference. Loosely described here, a servlet is application code transferred from a first server to a second for execution on the second server.

Because servers are typically accessed orders of magnitude more frequently than a client workstation, maintaining the integrity of the server executing a servlet becomes even more critical than maintaining the integrity of the client executing an applet. Corruption on a server can spread quite rapidly to any number of clients. Should corruption pass among servers, the rate of corruption of clients can increase exponentially. The sandbox of the servlet is appropriately restricted.

Accordingly, a server receiving a servlet over a network relies on the security aspects instantiated in the Java compiler, verifier and class loader as described generally below.

In the HotJava browser mentioned above, the boundaries of the sandbox for an applet are as follows: An applet may not read, write or inquire into the status of any filesystem on the client workstation **150**. An applet from an server, say, **120*b*** running on the workstation **150** can not access any other processor **120*a***, **120*n***, **150** over the network **160** other than its server processor **120*b***. An applet cannot load a library from either its server processor **120** or the workstation **150**. An applet cannot initiate the execution of a process. An applet cannot examine the properties of any resource on the workstation **150**.

To assist in the enforcement of the boundaries of the sandbox of an applet, the assignee of the instant invention has developed a suite of protocols for the development and execution of applets: the Java development environment (or Java Development Kit). The development environment includes a number of packages ("lang," "io," "net," "util," "awt" and "applet") To the extent an applet needs language, I/O, network, utility, windowing or application support, the application must resort to the methods available through the classes provided by one or some of the packages of the Java development environment.

Of course, not all classes and methods can be anticipated. As such, the Java development environment permits the construction of additional classes, methods and, ultimately, applets. However, the Java development environment includes a compiler which performs a number of checks to ensure that the applet does not contain any security violations. For example, the Java compiler does not permit pointers directly to memory. It strictly checks types. Access to an object must be through its public interface. The Java compiler leaves memory management for the Java interpreter, and the latter provides the former with no information on how it accomplishes the memory management

A Java verifier checks the code of an applet to ensure that the security of the Java development system is intact The verifier verifies the classfile, the type system checking, the bytecode and runtime type and access checking

After the verifier passes the applet code, the Java class loader checks the applet code to enforce namespaces based on the network source

Within each method of a package, the code uses the security manager to provide a system resource access control mechanism Any time an applet needs to access a particular resource, it uses a particular pre-defined method which provides checking as to whether the applet can access that particular resource.

In addition to the security aspects instantiated in the Java compiler, verifier and class loader as described above, the receiving server invokes an improved security manager according to the invention In addition to the security checks described above, the server's security manager identifies the network source of the servlet and implements a security policy based on the servlet's network source.

For example, the server's security manager may allow (or disallow) the execution of any servlet from a predetermined list of network sources. (In one embodiment, the authentication of the source of a servlet, particularly by digital signature, is the responsibility of the class loader.) Servlets from trusted servers identified by their digital signatures

would be executed. Alternatively, the server's security manager may allow (or disallow) the execution of any unsigned servlet As yet another alternative, the server's security manager may allow (or disallow) the execution of any signed servlet.

Another embodiment of the invention, including Java archives (JARs) is described in Table I

TABLE I

```
How to sign and verify JAR files
Benefit: signature is detached from .class file; and if servlet has only one class file, then signed
JAR file is just that .class file, plus the necessary signature info
Scenario for signing JAR files:
   servlet == list of files
To sign a servlet, use standalone Java program to
   1 specify list of files
   2 specify method (CA-based or PGP-style)
   3 create JAR file, create signature, write signed JAR file
The sun.server.util.jar class library includes support for doing the signing (as well as support for
packaging up the files into a JAR file.)
   1 Jar j = createJarFile (list of files)
   2 h = j .hash (algorithm)
   3 sign = h. encrypt (algorithm)
   4. SignedJARFile = createJarFile (j, sig, cert, alg)
Verifying signed servlet, within server classloader
   1 If (SignedJarFile)
     1 extract JarFile, sign, cert, alg from SignedJarFile
     2. c = JarFile.hash (algorithm) /* compute hash */
     3. pk = cert.getPublicKey ()
     4. h = sig.decrypt (algorithm)
     5 if (c == h) then /* signature is valid */
        is sig.ID on my list of trusted signatures?
        if yes, load servlet
A signed JAR file has those four fields.
The prototype uses PGP or X509 based certificate authorities.
SignedJarFile format
   * [Jar file | . class file]
   * signature
   * algorithm
   * [X509 certificate | PGP public key]
```

More generally, the server's security manager may decide whether to execute a servlet based on some other characteristic of the applet.

For such servlets as the server allows to execute, the server must also decide what server resources the servlet may access. The Java environment provides a default level of service However, the server can decide to enlarge or even shrink the default set of resources to which the servlet has access.

TABLE II

	read (filename)		socket (host, port)
	strings immutable	access authorization	DNS name resolution
server 120a	Yes	Yes	Yes
server 120b	No	No	No
. . .			
server 120n	Yes	No	Yes

In one embodiment of the invention, the server maintains a configurable security policy on a group or per-server basis (as described above) The server maintains a list of configurable resources, a list of the configurable accesses possible for each resource and a cross-list of servers (or groups of servers) and the accesses they may have. Table II illustrates one such configurable security policy wherein server 120*a* is not trusted at all and all security checks are applied to any servlet having server 120*a* as its source, while server 120 is sufficiently trusted not to need access authorization for read( )'s. Servlets from server 120*b* are completely trusted.

In the configurable policy embodiments, the namespace of the configurable embodiments supply the methods for the security manager's checks, typically by creating a subclass of the Java environment's security manager.

In one embodiment, the security manager includes a means for notifying the system administrator of the server that a servlet desires a resource from which it is currently excluded. On notification, the system administrator may reconfigure the security policy to allow the servlet the desired access. (Whether some class of servlets may cause notification of the administrator can also be configurable.)

In another embodiment, signed servlets are completely trusted (as server 120*b* in Table I) and have full access to the server. Unsigned servers, however, are blocked from executing HTTP requests and responses and inter-servlet communications. Unsigned servlets do not have access to the server's file system, properties files, dynamic configuration files, or memory management facilities.

FIG. 3 is a flowchart illustrating the flow of command according to an embodiment of the invention.

Of course, the client may likewise decide whether to execute application code (applet) loaded over the network based on the source or other characteristic of the applet.

Of course, the program text for such software as is herein disclosed can exist in its static form on a magnetic, optical or other disk, on magnetic tape or other medium requiring media movement for storage and/or retrieval, in ROM, in RAM, or in another data storage medium. That data storage medium may be integral to or insertable into a computer system.

What is claimed is:

1. In a client-server environment having a first server coupled to receive a program from a second server, a computer-implemented method for securing the execution of the program on said first server, said method comprising:

configuring said first server to permit execution of the program based on a configurable security characteristic of said program;

receiving at said first server said program transferred from said second server;

checking said program for said configurable security characteristic; and

executing said program on said first server if permitted by said configurable security characteristic and by a configurable security policy of said first server, said configurable security policy comprising a list of configurable resources, a list of configurable accesses possible for each resource, and a cross-list of servers and the configurable accesses the servers can have.

2. The method of claim 1 wherein said step of configuring comprises:

configuring said first server to permit the execution of a program based on its network source.

3. The method of claim 2 wherein said step of checking comprises checking for a digital signature.

4. The method of claim 2 wherein said step of configuring comprises:

configuring said first server to permit the execution of a program from any of a predetermined plurality of network sources.

5. The method of claim 2 wherein said step of configuring comprises:

configuring said first server to disallow the execution of a program from any of a predetermined plurality of network sources.

6. The method of claim 2 wherein said step of configuring comprises:

configuring said first server to allow the execution of only signed programs.

7. The method of claim 2 wherein said step of configuring comprises:

configuring said first server to allow the execution of unsigned programs.

8. The method of claim 1, further comprising:

rejecting pointers directly to memory above a predetermined level of said program.

9. The method of claim 1, further comprising:

strictly checking the usage of types in said program.

10. The method of claim 1 wherein said program comprises an object having a public interface, and the method further comprising the step of:

forcing access to said object through said public interface.

11. The method of claim 1 wherein before said step of executing the following step occurs:

verifying said program for security purposes.

12. The method of claim 11 wherein said step of verifying comprises

checking the format of the code of said program.

13. The method of claim 1 wherein before said step of executing the following step occurs:

ordering a plurality of classes according to the trustedness of the respective source of each class; and

rejecting a requested replacement of a class with a less trusted class.

14. The method of claim 1 wherein before said step of executing the following step occurs:

ordering a plurality of classes according to the trustedness of the respective source of each class; and

enforcing namespaces across said ordered plurality of classes.

15. The method of claim 1 wherein said step of executing comprises

accessing a resource by means of a predetermined method that checks whether said program can access said resource.

16. The method of claim 1 wherein said step of executing comprises accessing a resource, including checking said accessing against a configurable security policy.

17. The method of claim 16 wherein said step of accessing comprises

checking said accessing against a security policy configured on a per-resource basis.

18. The method of claim 16 wherein said step of accessing comprises

checking said accessing against a security policy configured on a access-type-per-resource basis.

19. The method of claim 16 wherein said step of accessing comprises

checking said accessing against a security policy configured to completely trust signed programs.

20. The method of claim 16 wherein said step of accessing comprises

checking said accessing against a securing policy configured to block unsigned programs from executing any one of Hypertext Transfer Protocol (HTTP) requests, HTTP responses and interservlet communications.

21. An article of manufacture comprising a medium for data storage wherein is located a computer program for causing a client-server computer system having a first server coupled to receive a program from a second server to secure the execution of a program on said first server processor by:

configuring said first server to permit the execution of the program based on a configurable security characteristic of said program transferred from said second server;

checking said program for said configurable security characteristic; and

executing said program on said first server if permitted by said configurable security characteristic and by a configurable security policy of said first server, said configurable security policy comprising a list of configurable resources, a list of configurable accesses possible for each resource, and a cross-list of servers and the configurable accesses the servers can have

22. A client-server computer system, comprising:

a network;

a first server; and

a second server, coupled to said first server by said network, said second server comprising a medium for data storage wherein is located a computer program for causing said second server to secure the execution of a program on said second server by:

configuring said second server to permit the execution of the program based on a configurable security characteristic of said program;

receiving said program from said first server;

checking said program for said configurable security characteristic; and

executing said program on said second server if permitted by said configurable security characteristic and by a configurable security policy of said second server, said configurable security policy comprising a list of configurable resources, a list of configurable accesses possible for each resource, and a cross-list of servers and the configurable accesses the servers can have.

23. A method for securing execution of a program on a client machine, comprising:

configuring the client machine to permit execution of the program based on a configurable security characteristic of the program;

providing the program to the client machine;

determining if the program contains the configurable security characteristic; and

executing the program based on the determination and based on a configurable security policy of a machine, said configurable security policy comprising a list of configurable resources, a list of configurable accesses possible for each resource, and a cross-list of machines and the configurable accesses the machines can have.

24. An article of manufacture specifying a representation of a program stored in a computer-readable storage medium and capable of execution by a machine in a distributed system, the article of manufacture comprising:

the program containing a configurable security characteristic in the computer-readable medium, the configurable security characteristic being detectable by the client machine and used by the client machine for determining whether to execute the program, wherein the client machine also executes the program based on a configurable security policy, said configurable security policy comprising a list of configurable resources, a list of configurable accesses possible for each resource, and a cross-list of machines and the configurable accesses the machines can have

25. In a client-server environment having a first server and a second server, a computer implemented method for securing execution of a program on said first server, said method comprising the steps of:

receiving the program from the second server;

executing the program on the first server in accordance with a configurable security policy; and

re-configuring the first server during execution of the program to reflect any changes to the configurable security policy determined during execution said configurable security policy comprising a list of configurable resources, a list of configurable accesses possible for each resource, and a cross-list of servers and the configurable accesses the servers can have

26. In a client-server environment having a first server coupled to receive a program from a second server, a computer-implemented method for securing execution of a servlet based on a characteristic of the servlet, said method comprising:

receiving at said second server a servlet having a configurable security characteristic; and

executing the servlet in accordance with the configurable security characteristic and a security policy associated with the second server, said configurable security policy comprising a list of configurable resources, a list of configurable accesses possible for each resource, and a cross-list of servers and the configurable accesses the servers can have.

* * * * *

EXHIBIT 15

US005983348A

United States Patent [19]

Ji

[11] **Patent Number: 5,983,348**

[45] **Date of Patent: Nov. 9, 1999**

[54] **COMPUTER NETWORK MALICIOUS CODE SCANNER**

[75] Inventor: Shuang Ji, Santa Clara, Calif.

[73] Assignee: Trend Micro Incorporated, Cupertino, Calif.

[21] Appl. No.: 08/926,619

[22] Filed: Sep. 10, 1997

[51] Int. Cl.^{6} G06F 13/00

[52] U.S. Cl. 713/200; 714/38

[58] Field of Search 395/186, 187.01, 395/188.01, 183.14, 183.13, 200.54, 200.55, 200.32; 380/3, 4, 23, 25; 713/200, 201, 202; 714/38, 37

[56] References Cited

U.S. PATENT DOCUMENTS

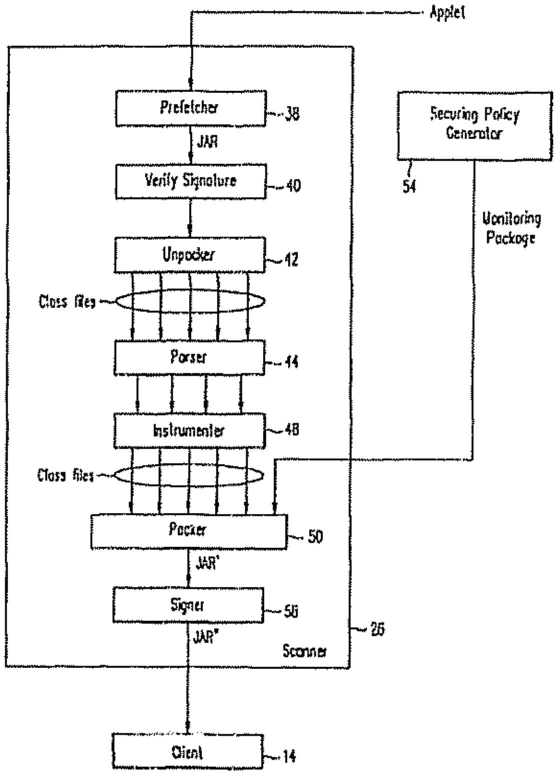
5,257,381	10/1993	Cook	395/700
5,359,659	10/1994	Rosenthal	380/4
5,390,232	2/1995	Freeman et al.	379/15
5,623,600	4/1997	Ji et al.	395/187.01
5,805,829	9/1998	Cohen et al.	395/200.32

Primary Examiner—Norman Michael Wright
Attorney, Agent, or Firm—Skjerven, Morrill MacPherson, Franklin & Friel LLP; Norman R. Klivans

[57] ABSTRACT

A network scanner for security checking of application programs (e.g. Java applets or Active X controls) received over the Internet or an Intranet has both static (pre-run time) and dynamic (run time) scanning. Static scanning at the HTTP proxy server identifies suspicious instructions and instruments them e.g. a pre-and-post filter instruction sequence or otherwise. The instrumented applet is then transferred to the client (web browser) together with security monitoring code. During run time at the client, the instrumented instructions are thereby monitored for security policy violations, and execution of an instruction is prevented in the event of such a violation.

34 Claims, 2 Drawing Sheets



Defendant's Trial Ex.
DTX - 1032
Case No. 06-369 GMS

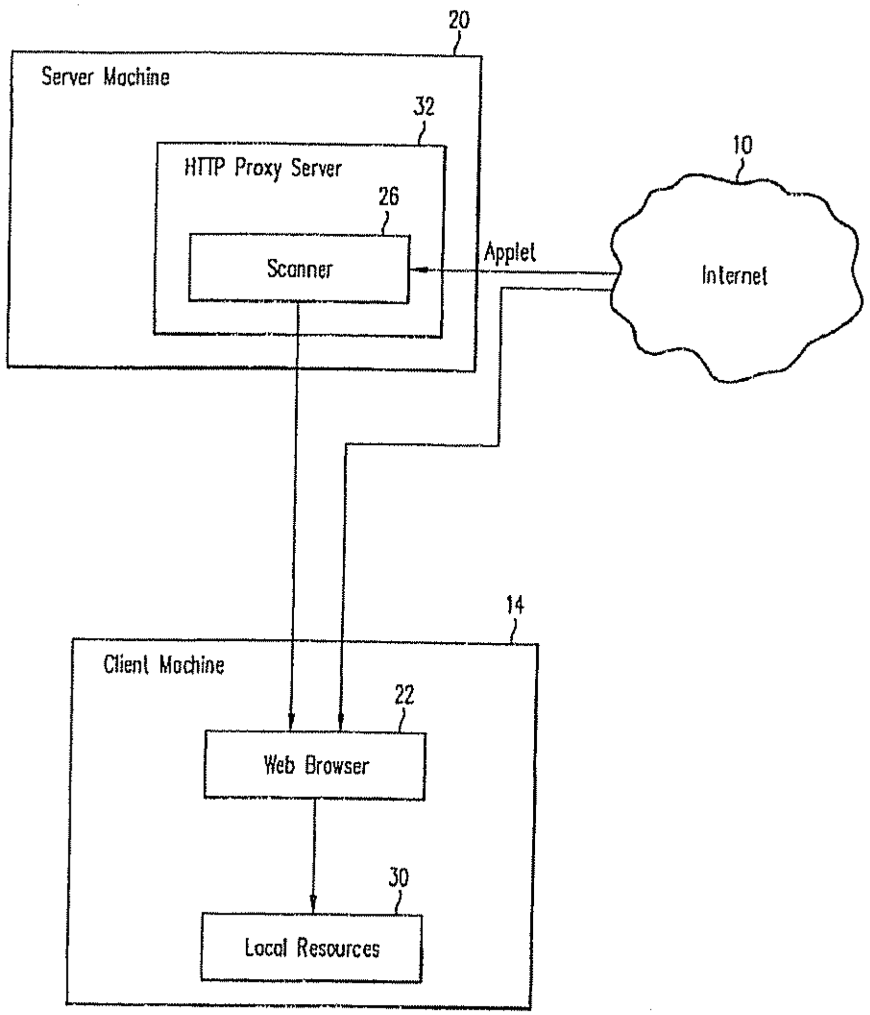

20
Server Machine
32
HTTP Proxy Server
26
Scanner
Applet
10
Internet
14
Client Machine
22
Web Browser
30
Local Resources

FIG. 1

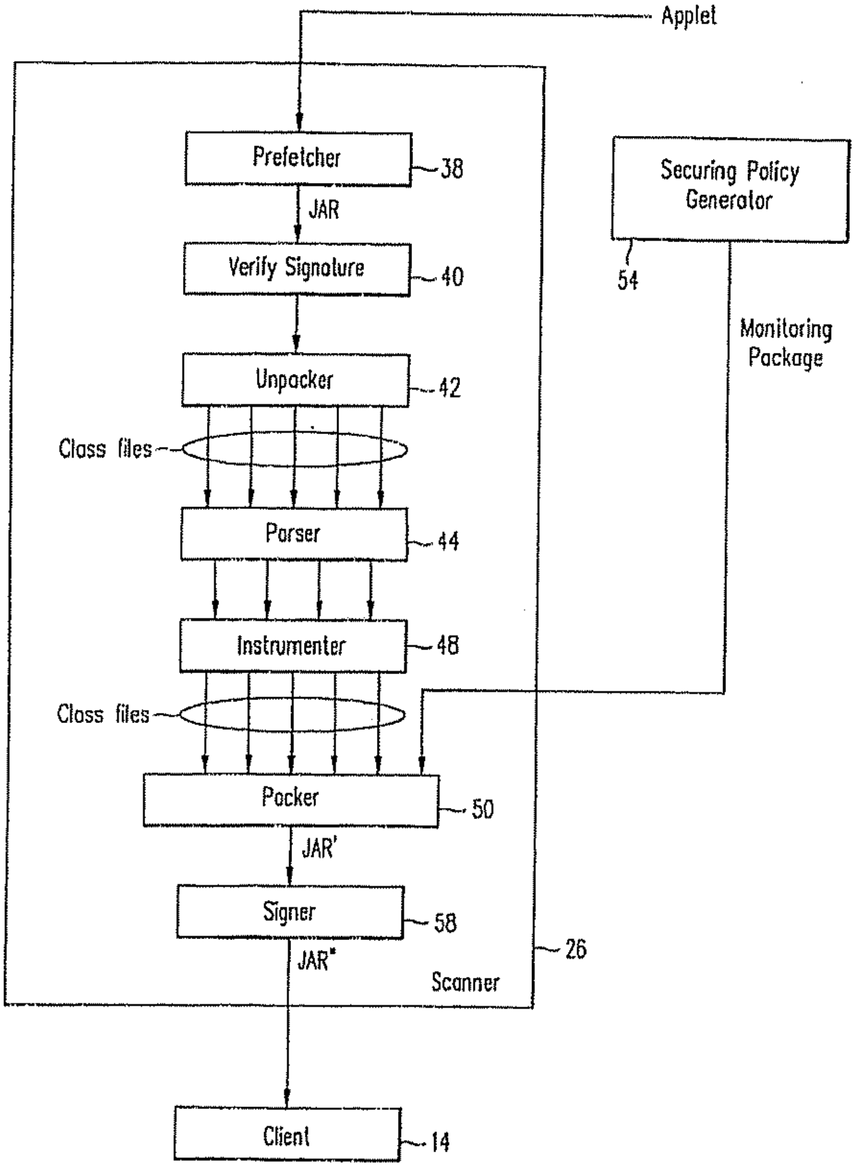

Applet
Prefetcher
38
JAR
Verify Signature
40
Unpacker
42
Class files
Parser
44
Instrumenter
48
Class files
Packer
50
JAR'
Signer
58
JAR"
Scanner
26
Client
14
Securing Policy Generator
54
Monitoring Package

FIG. 2

COMPUTER NETWORK MALICIOUS CODE SCANNER

FIELD OF THE INVENTION

This invention pertains to computer networks and specifically to detecting and preventing operation of computer viruses and other types of malicious computer code.

BACKGROUND

With the rapid development of the Internet, Intranet, and network computing, applications (application programs) are distributed more and more via such networks, instead of via physical storage media. Many associated distribution technologies are available, such as Java and Active X. Therefore objects with both data and code flow around the network and have seamless integration with local computer resources. However, this also poses a great security risk to users. Code (software) from unknown origin is thereby executed on local computers and given access to local resources such as the hard disk drive in a user's computer. In a world wide web browser environment, such code is often automatically executed and the user might not even have a chance to be forewarned about any security risks (e.g. presence of computer viruses) he bears. Attempts have been made to reduce such risks; see Ji et al., U.S. Pat. No. 5,623,600, incorporated by reference in its entirety.

Active X technology, like Java, distributes code that can access local system resources directly. The web browser cannot monitor or block such accesses. Such an applet (application) can do virtually anything that a conventional program, for instance, a virus, is capable of doing. Microsoft Corp. and others have attempted to address this problem by using digital signature technology, whereby a special algorithm generates a digital profile of the applet. The profile is attached to the applet. When an applet is downloaded from the Internet, a verification algorithm is run on the applet and the digital profile to ensure that the applet code has not been modified after the signing. If an applet is signed by a known signature, it is considered safe.

However, no analysis of the code is done to check the behavior of the applet. It is not difficult to obtain a signature from a reputable source, since the signature can be applied for online. It has occurred that a person has created an Active X applet that was authenticated by Microsoft but contains malicious code. (Malicious code refers to viruses and other problematic software. A virus is a program intended to replicate and damage operation of a computer system without the user's knowledge or permission. In the Internet/Java environment, the replication aspect may not be present, hence the term "malicious code" broadly referring to such damaging software even if it does not replicate.)

Java being an interpreted language, Java code can be monitored at run-time. Most web browsers block attempts to access local resources by Java applets, which protects the local computer to a certain extent. However, as the popularity of Intranets (private Internets) increases, more and more applets need to have access to local computers. Such restrictions posed by the web browsers are becoming rather inconvenient. As a result, web browsers are relaxing their security policies. Netscape Communicator is a web browser that now gives users the ability to selectively run applets with known security risks. Again, decisions are made based on trust, with no code analysis done.

Hence scanning programs with the ability to analyze and monitor applets are in need to protect users.

At least three Java applet scanners are currently available commercially: SurfinShield and SurfinGate, both from

Finjan, and Cage from Digitivity, Inc. SurfinShield is a client-side (user) solution. A copy of SurfinShield must be installed on every computer which is running a web browser SurfinShield replaces some of the Java library functions included in the browser that may pose security risks with its own. This way, it can trap all such calls and block them if necessary.

SurfinShield provides run-time monitoring. It introduces almost no performance overhead on applet startup and execution. It is able to trap all security breach attempts, if a correct set of Java library functions is replaced. However, it is still difficult to keep track of the states of individual applets if a series of actions must be performed by the instances before they can be determined dangerous this way, because the scanner is activated rather passively by the applets.

Since every computer in an organization needs a copy of the SurfinShield software, it is expensive to deploy. Also, installing a new release of the product involves updating on every computer, imposing a significant administrative burden.

Because SurfinShield replaces library functions of browsers, it is also browser-dependent; a minor browser upgrade may prevent operation. SufinGate is a server solution that is installed on an HTTP proxy server. Therefore, one copy of the software can protect all the computers proxied by that server. Unlike SufinShield, SurfinGate only scans the applet code statically. If it detects that one or more insecure functions might be called during the execution of the applet, it blocks the applet. Its scanning algorithm is rather slow. To solve this problem, SurfinGate maintains an applet profile database. Each applet is given an ID which is its URL. Once an applet is scanned, an entry is added to the database with its applet ID and the insecure functions it might try to access. When this applet is downloaded again, the security profile is taken from the database to determine the behavior of the applet. No analysis is redone. This means that if a previously safe applet is modified and still has the same URL, SurfinGate will fail to rescan it and let it pass through. Also, because the size of the database is ever-growing, its maintenance becomes a problem over time.

Cage is also a server solution that is installed on an HTTP proxy server, and provides run-time monitoring and yet avoids client-side installations or changes. It is similar to X Windows. All workstations protected by the server serve as X terminals and only provide graphical presentation functionality. When an applet is downloaded to Cage, it stops at the Cage server and only a GUI (graphical user interface) agent in the form of an applet is passed back to the browser. The applet is then run on the Cage server. GUI requests are passed to the agent on the client, which draws the presentation for the user. Therefore, it appears to users that the applets are actually running locally.

This approach creates a heavy load on the server, since all the applets in the protected domain run on the server and all the potentially powerful computers are used as graphical terminals only. Also, reasonable requests to access local resources (as in Intranet applications) are almost impossible to honour because the server does not have direct access to resources on individual workstations.

These products fail to create any balance between static scanning and run-time monitoring. SurfinShield employs run-time monitoring, SurfinGate uses static scanning, and Cage utilizes emulated run-time monitoring. Since static scanning is usually done on the server and run-time monitoring on the client, this imbalance also causes an imbalance between the load of the server and the client. To distribute the load between the client and the server evenly, the present inventor has determined that a combination of static scanning and run-time monitoring is needed.

SUMMARY

This disclosure is directed to an applet scanner that runs e.g. as an HTTP proxy server and does not require any client-side modification. The scanner combines static scanning and run-time monitoring and does not cause a heavy load on the server. It also does not introduce significant performance overhead during the execution of applets. The scanner provides configurable security policy functionality, and can be deployed as a client-side solution with appropriate modifications.

Thereby in accordance with the invention a scanner (for a virus or other malicious code) provides both static and dynamic scanning for application programs, e.g. Java applets or ActiveX controls The applets or controls (hereinafter collectively referred to as applets) are conventionally received from e.g. the Internet or an Intranet at a conventional server. At this point the applets are statically scanned at the server by the scanner looking for particular instructions which may be problematic in a security context. The identified problematic instructions are then each instrumented, e.g. special code is inserted before and after each problematic instruction, where the special code calls respectively a prefilter and a post filter. Alternatively, the instrumentation involves replacing the problematic instruction with another instruction which calls a supplied function.

The instrumented applet is then downloaded from the server to the client (local computer), at which time the applet code is conventionally interpreted by the client web browser and it begins to be executed. As the applet code is executed, each instrumented instruction is monitored by the web browser using a monitor package which is part of the scanner and delivered to the client side. Upon execution, each instrumented instruction is subject to a security check If the security policy (which has been pre-established) is violated, that particular instruction which violates the security policy is not executed, and instead a report is made and execution continues, if appropriate, with the next instruction.

More broadly, the present invention is directed to delivering what is referred to as a "live agent" (e.g., a security monitoring package) along with e.g. an applet that contains suspicious instructions during a network transfer (e.g. downloading to a client), the monitoring package being intended to prevent execution of the suspicious instructions. The suspicious instructions each may (or may not) be instrumented as described above; the instrumentation involves altering suspicious instructions such as by adding code (such as the pre-and post-filter calls) or altering the suspicious instructions by replacing any suspicious instructions with other instructions.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows diagramatically use of a scanner in accordance with this invention

FIG 2 shows detail of the FIG. 1 scanner.

DETAILED DESCRIPTION

Several characteristics of the well known Java language and applets are pertinent to the present scanning method and apparatus. Java is an interpreted, dynamic-linking language.

Only the application modules are distributed, and all the standard library functions are provided by the interpreter, for instance a web browser. Because Java byte code is platform-independent, applets have to use some of the standard library functions to access operating system resources.

This creates two opportunities in accordance with the invention to detect attempts to use operating system resources. First, one can "trick" applets into calling particular functions supplied by the scanner during the dynamic linking stage. This is done by replacing the browser Java library routines with the scanner's monitoring routines of the same name. Second, since invocations of such functions have to be resolved at run-time, symbolic names of these functions are kept in the Java applet module. The scanner can detect possible use of these functions by looking at the static code itself. The first opportunity provides run-time monitoring. It is the most definitive method to determine the security risks posed by an applet.

The second opportunity enables statically scanning an applet, without running it, to detect possible security risks. If a set of insecure functions is properly defined and an applet never calls any function in the set, the applet can be assumed to be safe. However, this static scanning method is not definitive, since an applet might show different behavior given different user input. Under certain conditions, the instruction in the applet that makes the function call may never be executed. If static scanning is used without run-time monitoring, many such "false alarms" of security risks are produced undesirably.

After the code of an applet is downloaded, e.g. via the Internet to a client platform (local computer), an instance of the applet is created in the conventional Java "virtual machine" in the web browser (client) running on that local computer. Different instances of the same applet might produce different results given different inputs. A running instance of an applet is conventionally called a session; sessions are strictly run-time entities. Static scanning cannot analyze sessions because static scanning does not let the applet run. Sessions are important because an instance of an applet will often perform a series of suspicious tasks before it can be determined dangerous (i.e., in violation of the security policy). Such state information needs to be associated with the sessions. The present applet scanner thereby stops sessions instead of blocking execution of the entire applet.

A security policy defines what functions an applet needs to perform to be considered a security risk. Examples of security policies include preventing(1) applets from any file access, or (2) file access in a certain directory, or (3) creating certain Java objects. An applet scanner in accordance with the invention may allow different security policies for different clients, for different users, and for applets from different origins.

FIG. 1 is a high level block diagram illustrating the present scanner in the context of conventional elements. The Internet (or an Intranet) is shown generally at 10. The client machine or platform (computer) 14, which is typically a personal computer, is connected to the Internet 10 via a conventional proxy server machine (computer) 20. Client machine 14 also includes local resources 30, e.g. files stored on a disk drive. A conventional web browser 22 is software that is installed on the client machine 14. It is to be understood that each of these elements is complex, but except for the presently disclosed features is conventional.

Upon receipt of a particular Java applet, the HTTP proxy server 32, which is software running on server machine 20 and which has associated scanner software 26, then scans the applet and instruments it using an instrumenter 28 which is part of the scanner software 26. (Downloaded non-applets are not scanned.) The instrumented applet is subject to a special digital signer which is an (optional) part of the scanner 26. The scanned (instrumented) applet, which has been digitally signed is then downloaded to the web browser 22 in the client 14. The applet is then conventionally interpreted by the web browser 22 and its instructions are executed. The execution is monitored by the monitor package software, also downloaded from scanner 26, in the web browser 22 in accordance with this invention for security purposes. Thus static scanning is performed by the HTTP proxy server 32 and dynamic scanning by the web browser 22.

The present applet scanner thus uses applet instrumentation technology, that is, for Java applets it alters the Java applet byte code sequence during downloading of the applet to the server 32. After the Java applet byte code sequence has been downloaded, the static (pre-run time) scanning is performed on the applet by the scanner 26. If an instruction (a suspicious instruction) that calls an insecure function (as determined by a predefined set of such functions) is found during this static scanning, a first instruction sequence (pre-filter) is inserted before that instruction and a second instruction sequence (post-filter) after that instruction by the instruments.

An example of such a suspicious Java function is "Java.IO.File.list" which may list the contents of a client (local) directory 30, e.g. a directory on the client machine 14 hard disk drive. The first instruction sequence generates a call to a pre-filter function provided by the scanner 26, signaling that an insecure (suspicious) function is to be invoked. The pre-filter checks the security policy associated with the scanner 26 and decides whether this particular instruction ("call") is allowed. The second instruction sequence generates a call to a post-filter function also provided by the scanner. It also reports the result of the call to the post-filter function. Both the pre- and post-filter functions update the session state to be used by the security policy. The static scanning and instrumentation are both performed on the HTTP proxy server 32.

The following is pseudo-code for the instrumentation process:

```
instrument (JavaClassFile classfile)
{
        extract constant pool from classfile;
        extract functions from classfile;
        for each function
{
        for each function
        {
        if( the instruction is a function call
             AND the target of the call is a pre-defined set
             of suspicious functions)
        {
        output an instruction sequence which generates a
        function call to a pre-monitor function, with the
        name of the suspicious function, parameters to the
        suspicious function, and possibly other
        information about his suspicious function
        invocation as the parameters;
        output the original instruction;
        output an instruction sequence which generates a
        function call to a post-monitor function, with the
        result of the suspicious function invocation and
        possibly other related information as parameters;
        }
```

-continued

```
        else
        {
        output the original instruction;
        }
      }
    }
}
```

Examples of pre- and post-monitor functions are:

(1) to disallow any directory listing access:

```
            pre-filter(function_name, parameters)
            {
            if (function_name == "Java.io.File.list")
            throw new SecurityException();
            }
            post-filter(result)
            {
            }
```

(2) To protect files under c:\temp from directory listing access:

```
    pre-filter(function_name, parameters)
    {
    if
    ((function_name == "java.io.File.list")
         {
         extract the name of the file to be read from
    parameters;
         if the directory to be listed is under c:\temp)
         throw new SecurityException();
    {
    }
    post-filter(result)
    {
    }
```

The pre and post filter and monitoring package security policy functions) are combined with the instrumented applet code in a single JAR (Java archive) file format at the server 32, and downloaded to the web browser 22 in client machine 14. From this point on, the server 32 is virtually disconnected from this server-client session. All the monitoring and applet code is executed in the web browser 22 in the client machine 14. The only time that the server 32 may be again involved during this particular session is when the applet is determined to be dangerous (i.e. including malicious code that violates the security policy) or the applet has completed execution, and a report is sent back to the server 32 by the monitoring code in the scanner 26. A report is optional in this second case.

This approach minimizes the overhead on both server 32 and browser 14. The only work performed on the server 32 is to identify suspicious applet instructions and instrument them, which is usually performed by a one time pass over the applet code. To the client web browser 22, the only overhead is some occasional calls to the scanner monitoring functions, which update session statistics and check security policies. This achieves an optimum distribution of scanning and monitoring between the server 32 and the client web browser 22. Also, the server 32 maintains no state information about active sessions in the set of host associated with the proxy server instead the session state information is maintained locally at client machine 14 by the downloaded monitoring functions.

This approach may damage the integrity of externally digitally signed (authenticated) applets, since the content of the applets is changed by the instrumentation. However, this can also be used as an advantage because using the present scanner, a new set of authenticated signatures can be set and enforced for the entire domain as further described below.

Operation of scanner 26 and its various (software) components is better understood with reference to FIG. 2, showing greater detail than FIG. 1.

An applet pre-fetcher component 38 fetches from the Internet 10 all the dependency files required by a Java class file, if they are not already packed into a JAR file. This is important because the goal is to attach the scanner monitor package to a session only once.

A Java applet may contain more than one code module, or class file. Heretofore this disclosure has assumed that all the class files are packed in one JAR file and downloaded once. One monitoring package is attached to the JAR file and every instantiation of this package on the client web browser 22 marks a unique session. However, if the class files are not packed together and are downloaded on an as-needed basis during applet execution, multiple instrumentation will occur and multiple instances of the monitoring package for the same session are created on the client. This creates a problem of how to maintain information on session states. To solve this problem, the pre-fetcher 38 pre-fetches the dependency class files during the static scanning of the main applet code module. The dependency class files are (see below) instrumented once, packed together, and delivered to the client.

Upon receiving a (signed) applet, the signal verifier component 40 then verifies the signature and its integrity, as conventional, to decide whether to accept this applet.

Next, the unpacker 42 component extracts the class files from the JAR file. JAR uses ZIP (compression) format.

Java class parser component 44 then parses each Java class file. Parser 44 conventionally extracts the instruction sequence of the Java functions.

The Java instrumenter component 48 instruments the Java class files, e.g. by inserting monitoring instructions (e.g. pre and post filter calls) before and after each suspicious instruction, as described above.

The monitor package contains monitoring functions that are delivered from the server 32 to the client web browser 22 with the instrumental applet and are invoked by the instrumentation code in the applet. The monitor package also creates a unique session upon instantiation. It also contains a security policy checker (supplied by security policy generator component 54) to determine whether the applet being scanned violates the security policy, given the monitoring information.

The security policy generator component 54 generates the security checker code included in the monitor package, from a set of predefined security policies. Different clients, users, and applets may have different security policies. The security policy generator 54 may run on server machine 20 or another computer. In addition, security policies can be configured by an administrator of the system. A simple security policy is to assign different weights to monitored functions and make sure the security weight of a session does not exceed a preset threshold. A more sophisticated security policy checks the file or resource the applet is trying to access at run time and prompts the user whether to allow the access. Hence the security policy broadly is a state machine to detect security policy violations upon attempted instruction execution.

The security policy generator 54 can operate outside the run-time instrumenter component 48 when the security policy is being created. The instrumenter component 48 can

then directly use the byte code. Thereby any performance limitations of the security policy generator component 54 become less important.

Next, packer 50 creates a new JAR file (JAR') from the instrumented class files and the monitoring package.

The digital signer component 58 digitally signs the applet (now JAR"), with a digital signature unique to the particular scanner 26, for authentication in the local domain. The applet JAR" is then transferred to the client machine 14 for execution. Thus the only signature that a client needs to recognize is the digital signature of the signer component 58 in the scanner 26. This pre-verification simplifies system administration and reduces risks to unsophisticated users who might otherwise accidentally accept applets with unauthorized signatures.

In one embodiment, the components of scanner 26 are each implemented in Java. Some (or all) of the functions ("components") of the scanner 26 described above may be implemented in native (non-Java) code to improve performance. The actual scanner code is not given here; it can be readily written by one of ordinary skill in the art in light of this disclosure.

This disclosure is illustrative and not limiting, further modifications will be apparent to one skilled in the art and are intended to fall within the scope of the appended claims.

I claim:

1. A method of detecting and preventing execution of instructions in an application program provided from a computer network, comprising:

providing the application program over the computer network;

determining whether the provided application program includes any instructions that are members of a particular set of instructions;

executing the application program if it is determined that no members of the set are included in the application program;

if it is determined that an instruction is a member of the set, then altering the application program, thereby allowing monitoring of execution of the instruction, wherein the altering includes inserting a first predefined call before the instruction and a second predefined call after the instruction; and

wherein the first or second predefined call changes a session state of the application program.

2. The method of claim 1, further comprising associating monitoring code with the application program.

3. The method of claim 1, wherein the altering includes replacing the instruction with a predefined second instruction.

4. The method of claim 1, wherein the application program is an applet.

5. The method of claim 4, wherein the applet is in the Java language.

6. The method of claim 1, wherein the computer network is an Intranet or the Internet.

7. The method of claim 1 wherein the first predefined call is a call to check a security policy.

8. The method of claim 7, wherein the security policy is a state machine.

9. The method of claim 1, wherein the inserting is repeated for each instruction in the application program that is a member of the particular set.

10. The method of claim 1, wherein the particular set of instructions includes instructions that access a predefined set of files

11. The method of claim 1, wherein the computer network includes a server and a client coupled to the server, and wherein the altering takes place at the server, wherein the executing the application program takes place at the client.

12. The method of claim 1, wherein the instructions are problematic instructions.

13. The method of claim 1, wherein the application program is altered at the point of the instruction.

14. A method of detecting and preventing execution of instructions in an application program provided from a computer network comprising:

- providing the application program over the computer network;
- determining whether the provided application program includes any instructions that are members of a particular set of instructions;
- executing the application program if it is determined that no members of the set are included in the application program;
- if it is determined that an instruction is a member of the set, then altering the application program, thereby allowing monitoring of execution of the instruction;
- determining if the application program includes an authentication;
- verifying the authentication; and
- replacing the verified authentication with a second authentication.

15. A method of detecting and preventing execution of instructions in an application program provided from a computer network, comprising:

- providing the application program over the computer network;
- determining whether the provided application program includes any instructions that are members of a particular set of instructions;
- executing the application program if it is determined that no members of the set are included in the application program;
- if it is determined that an instruction is a member of the set, then altering the application program, thereby allowing monitoring of execution of the instruction;
- providing all dependency files associated with the application program;
- providing a single monitoring package performing the step of determining for the application program and its associated dependency files; and
- executing the application program and its associated dependency files.

16. A scanner for detecting and preventing execution of instructions in an application program provided from a computer network, wherein the scanner determines whether the provided application program includes any instructions that are members of a particular set of instructions, allowing execution of the application program if it is determined that no members of the set are included in the application program; and comprising:

- an instrumenter which alters the application program at an instruction which is determined to be a member of the set, thereby allowing monitoring of execution of such instructions;
- wherein the instrumenter inserts a first predefined call before the instruction and a second predefined call after the instruction; and
- wherein the first or second predefined call changes a session state of the application program.

17. The scanner of claim 16, further comprising a packer which associates monitoring code with the application program.

18. The scanner of claim 16, wherein the instrumenter replaces the instruction with a predefined second instruction.

19. The scanner of claim 16, wherein the application program is an applet.

20. The scanner of claim 19, wherein the applet is in the Java language.

21. The scanner of claim 16, wherein the computer network is an Intranet or the Internet.

22. The scanner of claim 16, wherein the first predefined call is a call to check a security policy.

23. The scanner of claim 22, wherein the security policy is a state machine.

24. The scanner of claim 16, wherein the instrumenter repeats the inserting for each instruction in the application program that is a member of the particular set.

25. The scanner of claim 16, wherein the particular set of instructions includes instructions that access a predefined set of files.

26. The scanner of claim 16, wherein the computer network includes a server and a client coupled to the server, wherein the altering by the instrumenter takes place at the server, and wherein the executing the application program takes place at the client.

27. The scanner of claim 16, wherein the instructions that are members of the particular set are problematic instructions.

28. The scanner of claim 16, wherein the alteration is at the point of the instruction.

29. A scanner for detecting and preventing execution of instructions in an application program provided from a computer network, wherein the scanner determines whether the provided application program includes any instructions that are members of a particular set of instructions, allowing execution of the application program if it is determined that no members of the set are included in the application program; and comprising:

- an instrumenter which alters the application program at an instruction which is determined to be a member of the set, thereby allowing monitoring of execution of such instruction;
- a verifier which determines if the application program includes an authentication and verifies the authentication; and
- a signer which replaces the verified authentication with a second authentication.

30. A scanner for detecting and preventing execution of instructions in an application program provided from a computer network, wherein the scanner determines whether the provided application program includes any instructions that are members of a particular set of instructions, allowing execution of the application program if it is determined that no members of the set are included in the application program; and comprising:

- an instrumenter which alters the application program at an instruction which is determined to be a member of the set, thereby allowing monitoring of execution of such instruction;
- a prefetcher which fetches all dependency files associated with the application program; and
- a security policy generator which provides a single monitoring package for the application program and its associated dependency files.

31. A method of detecting and preventing execution of instructions in an application program provided from a computer network, comprising:

- providing the application program over the computer network;
- determining whether the provided application program includes any instructions that are members of a particular set of instructions;
- executing the application program if it is determined that no members of the set are included in the application program;
- if it is determined that an instruction is a member of the set, then altering the application program, thereby allowing monitoring of execution of the instruction;
- wherein the computer network includes a server and a client coupled to the server, and wherein the altering takes place at the server, wherein the executing the application program takes place at the client; and

performing the monitoring at the client.

32. A method of detecting and preventing execution of instructions in an application program provided from a computer network, comprising:

- providing the application program over the computer network;
- determining whether the provided application program includes any instructions that are members of a particular set of instructions;
- executing the application program if it is determined that no members of the set are included in the application program;
- if it is determined that an instruction is a member of the set, then altering the application program, thereby allowing monitoring of execution of the instruction, and
- carrying out the method for each of a plurality of application programs as each application program is provided from the computer network.

33. A scanner for detecting and preventing execution of instructions in an application program provided from a computer network, wherein the scanner determines whether the provided application program includes any instructions that are members of a particular set of instructions, allowing execution of the application program if it is determined that no members of the set are included in the application program; and comprising:

- an instrumenter which alters the application program at an instruction which is determined to be a member of the set, thereby allowing monitoring of execution of such instruction;
- wherein the computer network includes a server and a client coupled to the server, wherein the altering by the instrumenter takes place at the server, and wherein the executing the application program takes place at the client; and

wherein the monitoring is performed at the client.

34. A scanner for detecting and preventing execution of instructions in an application program provided from a computer network, wherein the scanner determines whether the provided application program includes any instructions that are members of a particular set of instructions, allowing execution of the application program if it is determined that no members of the set are included in the application program; and comprising:

- an instrumenter which alters the application program at an instruction which is determined to be a member of the set, thereby allowing monitoring of execution of such instruction; and
- wherein the instrumenter alters each of a plurality of application programs as each application program is provided from the computer network.

* * * * *

EXHIBIT 16

EXHIBIT 16 REDACTED

IN ITS ENTIRETY

EXHIBIT 17

EXHIBIT 17 REDACTED

IN ITS ENTIRETY

EXHIBIT 18

SECURE COMPUTING

+1 800.379.4944 toll free
+1 408 979 6572 International

Select Language
Search

Resources Products Try/Buy Support Partners

Locator

Home > Company > About Secure > Company Fact Sheet

Web Gateway | Messaging Gateway | Network Gateway | Identity and Access

Next Steps
- Evaluate Product

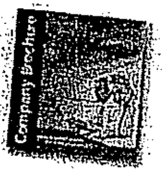
Company
About Secure
Company Fact Sheet
Executive Management
Board of Directors
Privacy Policy
Disclaimer
Press Center
Events / Seminars
Investor Relations
Careers
Contact Us

Testimonials

"Secure Computing® has become a case study for illustrating how quality leadership, investment in employees, and superior product design with unwavering focus on custom service, has resulted in increased market share, revenues, and profitability."

Jason Wright
Industry Analyst
Frost & Sullivan

Company Fact Sheet



Company overview

Secure Computing® is a global leader in Enterprise Gateway Security Solutions. Powered by our TrustedSource™ technology, our best-of-breed portfolio of solutions provides Web Gateway, Messaging Gateway, and Network Gateway security, as well as Identity and Access Management. Secure Computing is proud to be the security solutions provider to many of the most mission-critical and sensitive environments in the world. Please see below for more information on our solutions.

Our more than 19,000 customers, supported by a worldwide network of partners, include the majority of the Dow Jones Global 50 and the most prominent organizations in banking, financial services, healthcare, telecommunications, manufacturing, public utilities, and federal and local governments. With over 900 employees, the Company is headquartered in San Jose, California, and has offices worldwide.

Financial status	Market opportunity	Customers	Partners
Patents	Management team	Milestones	Solutions

Financial status

Secure Computing is publicly traded on the Nasdaq national Market System under the symbol SCUR. Annual revenues (reclassified for discontinued operations of the Advanced Technology contract revenues):

- 2006 Fiscal Revenues: $176.7M
- 2005 Fiscal Revenues: $109.2M
- 2004 Fiscal Revenues: $93.40M
- 2003 Fiscal Revenues: $76.21M
- 2002 Fiscal Revenues: $61.96M
- 2001 Fiscal Revenues: $48.35M
- 2000 Fiscal Revenues: $34.64M
- 1999 Fiscal Revenues: $22.54M

Defendant's Trial Ex.
DTX - 1296
Case No. 06-369 GMS

Market opportunity

Organizations today are expanding their businesses through the Internet daily. In this promising business environment, threats are also increasing right along with growth opportunities. Industry analyst IDC now expects the worldwide revenue for security hardware and software to be $30 billion by 2009.

Our customers need a secure infrastructure they can rely on. Accordingly, our commitment is to mitigate their risk exposure and protect their information assets from a multitude of threats, including identity theft, intruders, legal liability, security compromises, hackers, malicious software, and viruses.

Customers

Secure Computing's customers operate some of the largest and most sensitive networks and applications in the world. They include the majority of the Dow Jones Global 50 Titans and numerous organizations in the Fortune 1000, as well as banking, financial services, healthcare, telecommunications, manufacturing, public utilities, schools, and federal and local governments. Secure Computing has close relationships with the largest agencies of the United States government, including multiple contracts for advanced security research. Overseas, our customers are concentrated primarily in Europe, Japan, China, the Pacific Rim, and Latin America.

Partners

Our partnerships include a global network of OEMs, members of our Secure Alliance program, resellers, systems integrators, and companies that include our solutions in their product offerings. We offer our partners extensive support through our PartnersFirst Program and Web site. All business except for a select group of key accounts are sold through our partners. These companies include, for example and among others: Alternative Technology, Blue Coat Systems, Cisco, Computer

Associates, Comstor, Crossbeam, Dell, EDS, F5, Hewlett-Packard, McAfee, Microsoft, Network Appliance, NetOne Systems, Northrop Grumman, Novell, PGP Corporation, SafeNet, Sun PS, SAIC, Tech Data, Vertex Link, Voltage Security, Wavecrest Computing, Westcon, and Workshare.

Patents

Secure Computing is a leader in advanced research and development of network and systems security technology. Our team of distinguished researchers and scientists has achieved numerous breakthroughs in the security industry over the past 16 years. The company has been granted a total of 60 patents issued or pending in the United States. These patents cover systems architecture, cryptography, electronic mail filtering, and security control systems.

Management team

John McNulty — Chairman and Chief Executive Officer
Daniel Ryan — President and COO
Tim Steinkopf — Senior Vice President and Chief Financial Officer
Mike Gallagher — Senior Vice President, Product Development and Support
Mary K. Budge — Senior Vice President, Secretary, and General Counsel
Atri Chatterjee — Senior Vice President, Marketing

Secure Computing and Enterprise Gateway Security solutions

Secure Computing has long been regarded as the "gold standard" in Enterprise firewall technology. Our award-winning products protect some of the most mission-critical networks and applications in the world. As these network environments have evolved with the growth of the Internet, we have added extensive capabilities in the areas of Web and messaging application security as well as identity and access management. Addressing the fundamental issues of application security has required a new approach which is embodied in our Enterprise Gateway Security portfolio solutions listed below.

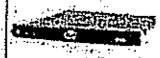
TrustedSource® - Global Intelligence to Protect Your Organization

TrustedSource technology is the most precise and comprehensive Internet host reputation system in the world, and a cornerstone of our solutions. TrustedSource uses data collected from over 7000 sensors worldwide to assign a reputation "score" to each sender encountered. This score is incorporated into Secure Computing products to enable them to quickly and accurately reject unwanted traffic.

Web Gateway Security Portfolio

Secure Computing offers a complete portfolio of Web Gateway Security appliances which protect enterprises from malware, data leakage, and Internet misuse, and ensure policy enforcement, regulatory compliance, and a productive application environment. Through our Trusted Source technology we are able to profile in real-time literally millions of entities connected to the Internet worldwide and provide up-to-the minute host behavior analysis to create a "reputation score" which can be used to determine whether a connection to the Enterprise network should be allowed to occur. We also employ the most sophisticated heuristic and signature-based techniques for stopping Malware as well as patented content analysis software for stopping data leakage.

Web Gateway Security product solutions.



Webwasher®

Webwasher® offers best-of-breed security solutions that protect both inbound traffic from all types of Web-borne threats (malware, viruses, spam, and spyware), and outbound traffic from data leakage of proprietary information.

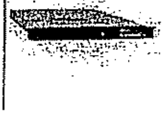
SmartFilter®

SmartFilter® enables organizations to understand and monitor their Internet use, while taking effective steps to provide appropriate control over outbound Web access.

Messaging Gateway Security Portfolio

Secure Computing is the global market leader in Messaging Gateway Security. We provide a complete portfolio of innovative, layered security solutions to stop inbound and outbound messaging threats in an integrated, best-of-breed, and technologically superior appliance. Secure Computing capabilities deliver maximum availability and unmatched security, effectiveness and global enterprise manageability across multiple messaging protocols including email, instant messaging, and Webmail. Secure Computing leverages this capability using global intelligence with our TrustedSource technology, providing the leading reputation system based upon real-time intelligence gathered from a worldwide network of thousands of sensors. The combination of easy-to-manage appliances with sophisticated centralized intelligence provides clean, efficient communications, eliminating both inbound and outbound risks.

Messaging Gateway Security product solutions.



IronMail®

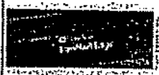
IronMail® delivers a centrally managed, integrated, best-of-breed messaging gateway security appliance for enterprises of all types and sizes.

IronIM™

The IronIM™ instant messaging security appliance is the first and only solution that integrates policy to secure, log, monitor and encrypt enterprise IM communications.

IronNet™

The IronNet™ appliance monitors all outbound Internet traffic, which is reviewed for compliance violations and subject to enforcement of corporate policies regarding compliance violations.



SecureEdge™

SecureEdge™ is a hardened appliance positioned at the perimeter of the mail system, applying TrustedSource technology to control email traffic at the network border.

RADAR™

RADAR

RADAR™ was developed to protect an organization's online reputation – whether by detecting and stopping Phishing scams or identifying and fixing PCs.

SECURE COMPUTING TrustedSource

TrustedSource

Secure Computing developed TrustedSource, the most precise and comprehensive Internet host reputation system in the world.

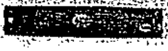
Network Gateway Security Portfolio

The definition of the "Enterprise Edge" has evolved significantly since the advent of the Web as mobile workforces, extranets, distributed applications, and an environment of highly sophisticated, blended threats has forced enterprises to deploy an array of security applications to provide services such as firewall, VPN, IDS/IPS, anti-virus, anti-spam and more.

As a pioneer in firewall technology and unified threat management, Secure Computing's application-layer products carry the highest possible Common Criteria certification and have never once been compromised in eleven years across thousands of deployments. Our ability to leverage TrustedSource technology to add real-time host profiling to our arsenal of security decision-making criteria makes Secure Computing's Network Gateway Security Portfolio truly unique.

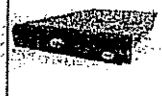
Network Gateway Security product solutions

SECURE COMPUTING Sidewinder



Sidewinder Network Gateway Security Appliance

Proven to be the most comprehensive security gateway appliance in the world, the Sidewinder® Security Appliance consolidates all major Internet security functions into a single system to defend against known and unknown threats.



CyberGuard TSP

TSP appliances are designed to protect mid-sized to large enterprises against both known and zero-hour attacks, using a hybrid architecture that combines stateful packet filtering, seven layer inspection, and secure content policy enforcement.

SnapGear Security Appliance

SnapGear™ is a complete office-in-a-box Internet security appliance for small businesses, with wide area networking tools normally only available in enterprise-class devices.

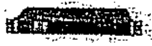
SECURE COMPUTING CommandCenter

CommandCenter

CommandCenter™ is Secure Computing's enterprise-class central management solution that enables you to implement security policies quickly, easily, and accurately across your entire security infrastructure.

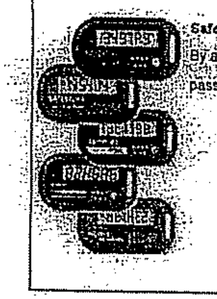
Identity and Access Management

Establishing the identity of users accessing corporate network resources, remotely and from the inside, as well as enforcing the policies governing the scope of these interactions, are critical functions of Enterprise Gateway Security. Accordingly, key components of our strategy are a dedicated Identity and Access Management appliance that combines strong authentication, remote access, and policy, and a wide variety of authentication form factors, including tokens.



SafeWord SecureWire

SafeWord® SecureWire™ is a powerful identity and access management (IAM) appliance that provides anywhere, anytime remote access to every application and data resource in the network for all remote and internal connections.



SafeWord

By authenticating users through VPNs, Citrix, dial-up, and Outlook Web Access, SafeWord® provides trusted access to corporate applications and networks from anywhere. SafeWord tokens deliver single-use passcodes, eliminating the vulnerabilities of passwords.

Security Services

Our award winning support team offers hands-on installation, training services, and 24x7 support.

www.securecomputing.com/goto/support

Milestones

- May 2007, Secure Computing Releases Next Generation of SnapGear SMB Network Security Appliance with TrustedSource
- May 2007, Secure Computing Launches Industry's First Self-Serve Domain Health Check
- April 2007, Secure Computing Awarded Three New Patents for Web and Messaging Gateway Security technologies
- March 2007, SC Magazine Bestows 'Best Buy' Honors on Secure Computing's IronMail
- March 2007, Secure Computing's Web Gateway Security Ranks #1 in Independent Office Documents Security Study
- February 2007, Secure Computing's Sidewinder Wins SC Magazine Reader's Trust Award for Best Enterprise Firewall
- January 2007, Sidewinder 7.0 and Webwasher 6.5 releases announced
- January 2007, Latest version of market-leading reputation system TrustedSource released
- January 2007, Secure Computing introduces Sidewinder 7.0
- January 2007, Webwasher 6.5 announced
- December 2006, Sidewinder G2 received Editor's Choice Award for functionality and security from Communications Week
- December 2006, Secure Computing honored with Best of 2006 awards for TSP 7300, SafeWord PremierAccess, and Webwasher
- December 2006, Secure Computing receives Reader Trust Finalist status from SC Magazine for Sidewinder G2, IronIM, and SmartFilter
- December 2006, Introduced the sleek, new carabiner-style Alpine token
- November 2006, Webwasher 6.0 released
- October 2006, Secure Computing positioned in the Leaders Quadrant for Gartner E-Mail Security Boundary 2006
- October 2006, Webwasher ranked the number 1 product for detecting the most malware by eWeek Magazine
- October 2006, Secure Computing holds Messaging Security Conference in Las Vegas
- September 2006, SnapGear awarded CRN Test Center's "Recommended" for Securing Wired and Wireless Access for SMEs
- September 2006, Secure Computing announces SafeWord SecureWire 50 IAM appliance for SMEs
- August 2006, Secure Computing closes acquisition of CipherTrust
- August 2006, Sidewinder G2 named Best MidMarket Product of the Year by CMP's VARBusiness Magazine
- July 2006, CyberGuard TSP achieves Common Criteria Certification using stringent DOD profile
- July 2006, Secure Computing announces intention to merge with CipherTrust, Leader in Messaging Security
- June 2006, Secure Computing named to FORTUNE Small Business Fastest-Growing Small Companies List
- June 2006, Secure Computing Positioned in Challengers Quadrant of Leading Analyst Firm's Magic Quadrant
- June 2006, Secure Computing Named to Business 2.0's 100 Fastest-Growing Technology Companies List
- June 2006, Announces Worldwide Channel Launch of Webwasher Secure Content Management Suite
- May 2006, IronMail named SC Magazine's Best Security Solution for Healthcare

- May 2006, Announces SafeWord PremierAccess 4.0
- May 2006, Sidewinder G2 Security Appliance Cryptographic Module for SecureOS Achieves FIPS 140-2 Validation
- May 2006, Secure Computing Honored with Reader Trust Awards from SC Magazine for Webwasher, SafeWord, SmartFilter
- April 2006, Extends TSP Portfolio to Support Unified Threat Management
- April 2006, Launches SafeWord SecureWire Identity and Access Management Appliance
- March 2006, Announces SnapGear Family of security appliances available to the Secure Computing worldwide channel
- March 2006, Secure Computing wins VARBusiness Magazine's 5-Star Rating 3 consecutive years
- February 2006, Information Security Magazine names IronMail 2006 Product of the Year
- February 2006, Announced Zero-hour Attack Protections technology inside Sidewinder G2
- January 2006, Acquires CyberGuard Corporation
- November 2005, Secure Computing reaches 1,000 Partner Milestone
- September 2005, Announces relationship with McAfee's line of SCM appliances to run SmartFilter technology
- August 2005, Celebrates Sidewinder G2 10-year flawless record of Zero compromises
- August 2005, Releases SmartFilter 4.1
- June 2005, Launches Sidewinder G2 Security Reporter
- May 2005, Sidewinder G2 first to achieve Common Criteria Certification using stringent DoD Application Firewall Medium Robustness Protection Profile
- April 2005, Secure Computing announces partnership with Sophos
- March 2005, Secure Computing wins VARBusiness Magazine's 5-Star Rating 2 consecutive years
- February 2005, SmartFilter wins SC Magazine Global Award
- February 2005, Announced 2005 new Sidewinder G2 lineup, including new SMB models
- December 2004, Sidewinder G2 named Product of the Year by Information Security Magazine
- September 2004, Secure Computing announces new version of SafeWord RemoteAccess 2.0
- September 2004, Launched Safeword PremierAccess 3.2
- September 2004, Launched SafeWord RemoteAccess, Cisco compatible
- August 2004, Sidewinder G2 Security Appliance receives EAL4+ Certification
- June 2004, Launched SmartFilter 4.0
- January 2004, Accelerated PartnersFirst Program by turning all but a select group of key accounts to channel partners
- January 2004, Announced Sidewinder G2 Security Appliance line
- November 2003, Announced SafeWord for Nortel Networks
- October 2003, Announced SafeWord for Check Point
- October 2003, Announced final acquisition of N2H2
- August 2003, SmartFilter receives OPSEC certification
- July 2003, Sidewinder G2 Firewall receives ICSA IPSec 1.1 certification
- April 2003, Sidewinder G2 Firewall achieves Common Criteria EAL4+ certification
- April 2003, Announced SafeWord for Citrix MetaFrame
- February 2003, Announced SmartFilter v3.2
- January 2003, Announced Sidewinder G2 Firewall and Sidewinder G2 Enterprise Manager
- January 2003, SmartFilter available on Cisco Routers
- January 2003, John McNulty named CEO of the year for Network Security by Frost & Sullivan
- June 2002, Unveiled next generation firewall plans
- May 2002, Announced SafeWord PremierAccess v3.1
- April 2002, Gauntlet awarded Common Criteria EAL4 level certification
- February 2002, Announced acquisition of Gauntlet firewall and VPN business
- January 2002, Announced Sidewinder Appliance
- December 2001, Announced SecureAlliance program
- November 2001, Sidewinder receives ICSA IPSec certification
- October 2001, Introduced SafeWord PremierAccess access control software for managing user authentication and authorization for e-Business applications
- July 2001, Sidewinder first firewall accepted into a Common Criteria Evaluation Assurance Level 4+
- April 2001, Introduced industry's first Embedded Firewall product
- April 2001, Announced $100,000 e-Security Challenge
- 1999, Sidewinder receives Firewall of the Year from Network Magazine

- 1995-1996, Acquired technology for two market leading access control products – SafeWord® authentication, and SmartFilter® URL filtering
- 1995, Announced Sidewinder challenge
- 1995, Introduced the world's first truly secure proxy-based firewall, Sidewinder®
- 1995, Initial Public Offering
- 1989, Spun off as Secure Computing Corporation
- 1984, Secure Computing started as the Secure Computing Technology Center division of Honeywell

Secure Computing is a global leader in Enterprise Gateway Security software solutions. Powered by our TrustedSource technology which provides real-time web and messaging reputation scoring, our award winning portfolio of email, Web, and application firewall security solutions provide anti-spam, anti-virus, anti-phishing, anti-malware, and anti-spyware prevention and protection to help ensure enterprise network security. Secure Computing's security software and network appliances also provide data leakage prevention, regulatory compliance, including robust auditing and reporting, strong authentication, and identity management.

[illegible] Control | Anti-malware | Anti-phishing | Anti-spam | Anti-spyware | Anti-virus | Application Firewall | Auditing & Reporting | Authentication | CIPA Compliance [illegible]
[illegible] Content Filtering | Data Leakage | Email Security | Enterprise [illegible] | Global Intelligence | Identity Management |
Internet Security | Internet Security Solutions | Intrusion [illegible] | Messaging Security | Network Gateway Security
Network Management | Network Security | Network security software | Online Banking | [illegible] | [illegible] Authentication | Regulations Compliance [illegible]
[illegible] | Reputation System | Security Appliance | Security audit | [illegible] Software | Spam Blocker | Spam Filter | Spam [illegible]
[illegible] | TrustedSource | Unified Threat Management | [illegible] | Virus Signature | VPN | Web 2.0 Threats [illegible]
| Web Gateway Security | Web [illegible] Network Security

EXHIBIT 19

REDACTED

CyberGuard Corporation
Budget vs. Actual
For the Year Ended June 30, 2005

Consolidated WebWasher	Actual Q1	Actual Q2	Actual Q3	Actual Q4
Revenues Products				
Software				
Americas:				
North America	641,898	881,418	647,076	729,634
Federal	-	-	-	-
Latin America	-	-	-	-
Europe:	-	-	-	-
United Kingdom	293,174	218,736	158,594	347,178
EMEA (Excluding UK)	1,602,945	2,382,844	1,998,059	2,864,803
Asia	75,788	148,138	191,144	144,173
Total Software	2,613,806	3,631,136	2,994,873	4,085,788
Total Product Revenue	2,613,806	3,631,136	2,994,873	4,085,788
Services:				
Maintenance				
Americas:				
North America	96,403	96,025	124,570	121,187
Federal	-	-	-	-
Latin America	-	-	-	-
Europe:	-	-	-	-
United Kingdom	19,940	22,749	72,428	52,410
EMEA (Excluding UK)	168,820	239,613	255,037	294,379
Asia	15,771	13,702	17,532	15,095
Total Maintenance	300,934	372,089	469,567	483,071
Total Service Revenue	300,934	372,089	469,567	483,071
Total Revenues	2,914,739	4,003,225	3,464,440	4,568,859
Cost Of Sales				
Software				
Acquisition Costs	408,363	408,363	408,363	408,363
Americas:	-			
North America	8,817	166,275	96,951	80,809
Federal	-	-	-	-
Latin America	-	-	-	-
Europe:	-	-	-	-
United Kingdom	34,437	40,533	55,248	46,046
EMEA (Excluding UK)	354,303	448,408	441,674	479,599
Asia	12,526	13,746	52,955	15,374
Total Software	410,083	668,963	646,828	621,828
Total Product Cost of Sales	818,446	1,077,326	1,055,191	1,030,191
Services:				
Worldwide Support	-	-	-	-
Maintenance				
Americas:				

REDACTED

HIGHLY CONFIDENTIAL-
ATTORNEYS' EYES ONLY

HIGHLY CONFIDENTIAL-
ATTORNEYS' EYES ONLY

REDACTED

REDACTED

EXHIBIT 20

6/20/05

By Product Line (FY 2005)

	Q1	Q2	Q3	Q4	FY
CYG	4,378,010	4,371,166	4,444,208	5,568,691	18,762,076
SG	2,331,913	1,866,718	1,235,770	1,532,057	6,966,459
WW	738,301	977,443	771,646	850,821	3,338,212
NA Total	7,448,224	7,215,328	6,451,625	7,951,569	29,066,747
CYG	580,777	322,019	678,281	686,607	2,267,684
CUK	3,071,766	3,839,756	3,851,427	2,964,738	13,727,687
SG	459,155	334,946	312,801	271,232	1,378,133
WW	2,084,879	2,863,941	2,484,118	3,558,771	10,991,710
EMEA	6,196,578	7,360,662	7,326,627	7,481,348	28,365,214
CYG	1,518,355	1,284,088	2,050,492	1,189,506	6,042,441
SG	424,539	482,456	667,001	428,620	2,002,616
WW	91,559	161,840	208,676	159,267	621,342
APAC	2,034,453	1,928,385	2,926,169	1,777,393	8,666,399
TOTAL	15,679,255	16,504,375	16,704,420	17,210,310	66,098,360

Defendant's Trial Ex.
DTX - 1320
Case No. 06-369 GMS

HIGHLY CONFIDENTIAL-
ATTORNEYS' EYES ONLY

H:\M&A\CyberGuard\Quarterly Results FY 2005.xls

HIGHLY CONFIDENTIAL-
ATTORNEYS' EYES ONLY

CyberGuard Corporation
Budget vs. Actual
For the Year Ended June 30, 2005

	Actual Jul	Actual Aug	Actual Sep	Actual Oct	Actual Nov	Actual Dec	Actual Jan	Actual Feb	Actual Mar	Actual Apr	Actual May	Actual Jun	Actual Q1	Actual Q2	Actual Q3	Actual Q4
Consolidated WebWasher																
Revenues Products ·																
Software																
Americas:																
North America	331,895	143,974	166,029	165,207	139,351	576,861	718,585	199,537	228,954	153,619	164,558	411,457	641,898	881,418	647,076	729,634
Federal													-	-	-	-
Latin America													-	-	-	-
Europe:													-	-	-	-
United Kingdom	108,498	144,063	40,613	45,138	41,583	132,014	45,519	35,669	77,406	72,155	67,210	207,813	293,174	218,736	158,594	347,178
EMEA (Excluding UK)	372,975	488,456	741,514	409,382	866,849	1,106,613	304,409	471,440	1,222,210	187,415	706,611	1,970,777	1,602,945	2,382,844	1,998,059	2,864,803
Asia	30,447	19,429	25,912	42,271	49,291	56,576	48,139	20,664	122,341	36,049	22,577	85,547	75,788	148,138	191,144	144,173
Total Software	843,815	795,923	974,068	661,998	1,097,074	1,872,065	616,652	727,310	1,650,911	449,238	960,956	2,675,594	2,613,806	3,631,136	2,994,873	4,085,788
Total Product Revenue	843,815	795,923	974,068	661,998	1,097,074	1,872,065	616,652	727,310	1,650,911	449,238	960,956	2,675,594	2,613,806	3,631,136	2,994,873	4,085,788
Services:																
Maintenance																
Americas:																
North America	29,356	28,894	38,153	30,269	31,015	34,741	35,862	37,749	50,960	33,720	41,829	45,638	96,403	96,025	124,570	121,187
Federal													-	-	-	-
Latin America													-	-	-	-
Europe:													-	-	-	-
United Kingdom	6,522	6,484	6,933	7,256	8,917	6,576	8,766	9,914	53,749	12,264	11,925	28,222	19,940	22,749	72,428	52,410
EMEA (Excluding UK)	53,032	58,824	56,965	64,280	94,487	80,846	80,466	86,909	87,661	86,544	86,205	121,630	168,820	239,613	255,037	294,379
Asia	5,073	5,453	5,246	4,427	4,593	4,682	6,849	5,282	5,400	5,391	5,192	4,512	15,771	13,702	17,532	15,095
Total Maintenance	93,983	99,654	107,296	106,232	139,012	126,845	131,943	139,854	197,770	137,919	145,151	200,001	300,934	372,089	469,567	483,071
Total Service Revenue	93,983	99,654	107,296	106,232	139,012	126,845	131,943	139,854	197,770	137,919	145,151	200,001	300,934	372,089	469,567	483,071
Total Revenues	937,798	895,577	1,081,365	768,230	1,236,085	1,998,910	748,595	867,164	1,848,681	587,157	1,106,107	2,875,595	2,914,739	4,003,225	3,464,440	4,568,859
Cost Of Sales																
Software																
Acquisition Costs	136,121	136,121	136,121	136,121	136,121	136,121	136,121	136,121	136,121	136,121	136,121	136,121	408,363	408,363	408,363	408,363
Americas:																
North America	28,986	(44,770)	24,601	27,780	22,520	115,975	49,816	23,930	23,205	25,113	26,478	29,219	8,817	166,275	96,931	80,809
Federal													-	-	-	-
Latin America													-	-	-	-
Europe:													-	-	-	-
United Kingdom	56,996	11,788	(34,347)	13,344	12,976	14,213	29,785	12,056	13,407	12,259	11,496	22,291	34,437	40,533	55,248	46,046
EMEA (Excluding UK)	61,416	127,196	165,692	138,183	150,434	159,792	101,863	108,833	230,978	125,730	62,614	291,255	354,303	448,408	441,674	479,599
Asia	4,520	3,525	4,481	2,913	8,598	2,236	9,842	3,275	39,837	8,715	1,694	4,964	12,526	13,746	52,955	15,374
Total Software	151,917	97,739	160,427	182,219	194,527	292,216	191,306	148,095	307,427	171,817	102,282	347,730	410,083	668,963	646,828	621,828
Total Product Cost of Sales	288,038	233,860	296,548	318,340	330,648	428,337	327,427	284,216	443,548	307,938	238,403	483,851	818,446	1,077,326	1,055,191	1,030,191
Services:																
Worldwide Support																
Maintenance													-	-	-	-
Americas:																
North America	6,917	8,320	17,553	6,361	12,036	8,383	7,374	14,175	17,482	20,943	25,110	14,625	32,790	26,779	39,031	60,678

H:\M&A\CyberGuard\Quarterly Results FY 2005.xls

Consolidated WebWasher	Actual Jul	Actual Aug	Actual Sep	Actual Oct	Actual Nov	Actual Dec	Actual Jan	Actual Feb	Actual Mar	Actual Apr	Actual May	Actual Jun	Actual Q1	Actual Q2	Actual Q3	Actual Q4
Federal													-	-	-	-
Latin America													-	-	-	-
Europe:													-	-	-	-
United Kingdom	1,537	1,867	3,190	1,525	3,460	1,587	1,803	3,723	18,438	7,617	7,159	9,044	6,594	6,572	23,964	23,819
EMEA (Excluding UK)	12,495	16,938	26,208	13,508	36,668	19,507	16,546	32,635	30,072	53,751	51,749	38,978	55,641	69,683	79,253	144,478
Asia	1,195	1,570	2,413	930	1,782	1,130	1,408	1,984	1,853	3,348	3,117	1,446	5,179	3,843	5,244	7,911
Total Maintenance	22,144	28,694	49,365	22,324	53,947	30,606	27,131	52,516	67,845	85,659	87,134	64,092	100,204	106,876	147,492	236,885
Total Service Cost of Sales	22,144	28,694	49,365	22,324	53,947	30,606	27,131	52,516	67,845	85,659	87,134	64,092	100,204	106,876	147,492	236,885
Total Cost of Sales	310,182	262,554	345,913	340,664	384,595	458,943	354,558	336,732	511,393	393,597	325,537	547,943	918,650	1,184,202	1,202,683	1,267,077
Gross Profit	627,616	633,022	735,451	427,566	851,490	1,539,967	394,037	530,432	1,337,288	193,560	780,570	2,327,652	1,996,069	2,819,023	2,261,757	3,301,783
Gross Margin	66.92%	70.68%	68.01%	55.66%	68.89%	77.04%	52.64%	61.17%	72.34%	32.97%	70.57%	80.95%	68.48%	70.42%	65.28%	72.27%
Operating Expenses:																
Sales & Marketing	458,885	571,840	646,786	515,343	865,776	660,638	521,297	474,814	700,510	469,083	557,626	1,501,234	1,677,511	2,041,758	1,696,620	[illegible]
Development	124,872	113,219	124,022	120,163	117,717	118,956	154,815	156,786	128,765	117,913	98,358	63,414	362,112	356,836	440,386	[illegible]
General & Administrative	146,226	141,653	164,954	180,425	188,852	35,509	154,221	183,970	145,601	159,308	183,333	361,975	452,833	404,786	483,792	[illegible]
Total Operating Expenses	729,983	826,711	935,762	815,931	1,172,346	815,103	830,333	815,569	974,896	746,304	839,317	1,926,622	2,492,456	2,803,380	2,620,799	3,512,243
Operating Profit	(102,367)	(193,689)	(200,311)	(388,365)	(320,855)	724,863	(436,296)	(285,137)	362,392	(552,744)	(58,747)	401,030	(496,367)	15,643	(359,042)	(210,461)
Operating Margin	-10.92%	-21.63%	-18.52%	-50.55%	-25.96%	36.26%	-58.28%	-32.88%	19.60%	-94.14%	-5.31%	13.95%	-17.33%	1.16%	-19.24%	-28.31%
Other Income(Expense)																
Amortization Expense	(140,777)	(67,297)	(140,715)	(140,715)	(140,830)	(140,909)	(140,821)	(140,384)	(140,847)	(140,750)	(141,428)	(140,482)	(348,789)	(422,454)	(422,051)	(422,660)
Software Amortization Expense	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-
Depreciation Expense	(13,716)	(13,393)	(13,394)	(13,761)	(14,049)	(14,337)	(13,840)	(13,690)	(14,095)	(13,433)	(13,996)	(13,317)	(40,503)	(42,146)	(41,625)	(40,746)
Currency Exchange Gain/Loss	(3,854)	(2,938)	(36,152)	(7,912)	(40,357)	6,702	19,673	(19,522)	28,572	(736)	7,508	28,752	(42,945)	(41,567)	28,723	35,523
Gain/Loss on Sale of Fixed Assets	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-
Miscellaneous Income	12,364	9,509	11,205	13,993	20,537	8,618	11,897	11,851	6,915	7,873	7,526	52,182	33,078	43,148	30,664	67,580
Miscellaneous Expense	(1,298)	0	(55)	(19,633)	(6,904)	(1,575)	(1,097)	(9,351)	(160,913)	(185)	(132)	100,200	(1,353)	(28,113)	(171,360)	99,883
Total Other Income(Expense)	(147,281)	(74,119)	(179,111)	(168,029)	(181,603)	(141,500)	(124,188)	(171,096)	(280,367)	(147,231)	(140,523)	27,335	(400,511)	(491,131)	(575,650)	(260,419)
Interest Income(Expense)																
Interest Income	213	0	1,392	160	0	301	0	0	0	332	0	0	1,606	461	-	332
Interest Expense - Convertible Debt	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-
Interest Expense - Warrants	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-
Interest Expense - Other	(412)	(559)	(13)	(74)	(18)	(121)	0	(101)	0	0	0	0	(984)	(213)	(101)	-
Total Interest Income(Expense)	(199)	(559)	1,379	87	(18)	180	-	(101)	-	332	-	-	622	249	(101)	332
Net Income(Loss) Before Tax	(249,847)	(268,366)	(378,043)	(556,307)	(502,476)	583,543	(560,484)	(456,334)	82,025	(699,644)	(199,270)	428,365	(896,256)	(475,240)	(934,792)	(470,549)
Provisions for Income Tax()																
Current	0	0	0	0	0	0	0	0	0	0	0	(41,072)	-	-	-	(41,072)
Deferred	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-
Total Provisions for Income Tax	-	-	-	-	-	-	-	-	-	-	-	(41,072)	-	-	-	(41,072)
Net Income After Tax	(249,847)	(268,366)	(378,043)	(556,307)	(502,476)	583,543	(560,484)	(456,334)	82,025	(699,644)	(199,270)	387,293	(896,256)	(475,240)	(934,792)	(511,621)
Net Profit Margin	-26.64%	-29.97%	-34.96%	-72.41%	-40.65%	29.19%	-74.87%	-52.62%	4.44%	-119.16%	-18.02%	13.47%	-30.75%	-11.87%	-26.98%	-11.20%

H:\M&A\CyberGuard\Quarterly Results FY 2005.xls

HIGHLY CONFIDENTIAL-
ATTORNEYS' EYES ONLY

EXHIBIT 21

Secure Computing Corporation
Gross Margin Report
Product: Webwasher
For The Month Ending December 31, 2006

	2005 Q4 QTR	% OF REVENUE	2006 Q1 QTR	% OF REVENUE	2006 Q2 QTR	% OF REVENUE	2006 Q3 QTR	% OF REVENUE	2006 Q4 QTR	% OF REVENUE	2007 Q1 QTR	% OF REVENUE	2007 Q2 QTR	% OF REVENUE
Revenue														
Software	$0	0%	$0	0%	$90,499	2%	$45,764	2%	$92,274	2%	$59,225	1%	$149,798	2%
Appliance	0	0%	0	0%	453,864	10%	273,122	9%	294,400	7%	600,244	14%	724,146	9%
Installations	0	0%	0	0%	0	0%	0	0%	6,810	0%	17,274	0%	41,946	1%
Training - Security Services	0	0%	0	0%	26,055	1%	29,393	1%	25,587	1%	45,391	1%	41,467	1%
12 Month Subscriptions	0	0%	2,315,527	94%	2,401,137	52%	2,602,017	86%	2,858,296	66%	2,890,690	71%	3,093,997	40%
24 Month Subscriptions	0	0%	30,099	1%	41,160	1%	50,925	2%	115,121	3%	148,220	3%	161,657	2%
36 Month Subscriptions	0	0%	109,616	4%	158,899	3%	169,118	6%	200,188	6%	584,910	13%	522,205	7%
Royalty Revenue	0	0%	0	0%	1,437,342	31%	(154,052)	(5%)	651,271	15%	(220,914)	(5%)	306,035	38%
VSOE Bundled Elements											28,621	1%	76,688	1%
Total Revenue·	0	0%	2,455,242	100%	4,608,958	100%	3,016,287	100%	4,323,927	100%	4,231,861	100%	5,118,139	100%
Total Cost of Goods Sold	0	0%	744,585	30%	988,173	21%	929,479	31%	718,286	17%	1,176,001	28%	662,561	9%
Gross Margin (Non-GAAP)	0	0%	1,710,657	70%	3,620,783	79%	2,086,808	69%	3,605,641	83%	3,055,860	72%	4,455,578	91%
Operating Expense:														
S&M			1,031,202		1,935,762		1,266,841		1,816,049		1,777,382		2,149,618	
R&D			343,734		645,254		422,280		605,350		677,098		870,084	
G&A			122,762		230,448		150,814		216,196		211,593		255,907	
Total Operating Expense			1,497,698		2,811,463		1,839,935		2,637,595		2,666,072		3,275,609	
Net Income/(loss)			212,959	9%	809,320	18%	246,873	8%	968,046	22%	389,788	9%	1,179,969	23%

H:\M&A\CyberGuard\Webwasher GM 2006 and beyond.xls

Defendant's Trial Ex.
DTX - 1321
Case No. 06-369 GMS

HIGHLY CONFIDENTIAL-
ATTORNEYS' EYES ONLY

EXHIBIT 22

From:
To:
CC:
BCC:
Sent Date: 0001-01-01 00:00:00:000
Received Date: 0001-01-01 00:00:00:000
Subject:
Attachments:

Product Meeting Minutes

Date: June 1, 2004
Participants: Benita, Frank, Heiko, Martin, Peter, Roland, Thomas, Tom
Minutes: Roland Cuny

Agenda:

1. Webwasher 5.0.2 (Update)
2. Webwasher 5.0.1 (Update)
3. Webwasher 5.1 (Planning)
4. OEM Support Responsibilities (Anti Virus)
5. Webwasher Appliance (CyberGuard)
6. T-Online (Anti Spam)
7. Extranet
8. Product Marketing Manager
9. Content Reporter
10. Competition Paper

1. Webwasher 5.0.2 (Update)
- Most features requested from customers were implemented. 'Digest Function' is in works. Release is on track as planned (mid or end of June).

2. Webwasher 5.0.1 (Update)
- Customers complain about poor stability of the software. Problems are not severe crashes but tedious bugs which hinder productivity. As a consequence development will dedicate 80% of this week's time for bug fixing and maintenance.
- NetCologne received a stable new version which should fix their problem. Next week, Frank and Jürgen will visit the customer to improve confidence.

3. Webwasher 5.1 (Planning)
- There was a meeting to discuss potential implementation methods for the proactive security feature (aka 'Finjan-Killer'). Two solutions were elaborated. First, we could copy Finjan's features. This idea was dropped because the gain in security is questionable and development is too time consuming. Second, we develop our own mix of methods which are more favorable for corporate customers needs. A stricter policy will block anything that is not proven to be harmless. Several implementation ideas were developed which need further discussion, e.g. sandboxing of javascript or verification of applet certificates.

Defendant's Trial Ex.
DTX - 1056
Case No. 06-369 GMS

- The important CyberGuard integration topics drain too many development resources. 10-12 MW are missing (QA not included). This requires further actions.

4. OEM Support Responsibilities (Anti Virus)
- We lack a workflow process to handle complains of customers regarding malicious code. Frank to arrange meeting with Peter, Roland and Support.

5. Webwasher Appliance (CyberGuard)
- CG Linux ('CyberGuard Linux' = hardened Red Hat Linux 8) was selected as the operating system of choice for the appliance. Hence this is actually another integration platform we have to support.
- The appliance will integrate five CSM products (no CR, no IM)
- Need to solve Red Hat 8 multiprocessor problems, asap.
- Hardware has to be sent to Paderborn asap to integrate our software, run performance benchmarks and finally to deliver an image to the hardware manufacturer
- Release scheduled as follows:

Appliance prototype phase (mock up, housing +GUI): End of June
Appliance beta phase (functional for beta customers): Mid/End June
Appliance FCS (first customer shipment): 2nd half of August

- Open issue of adapting the GUI to CyberGuard look&feel.

6. T-Online (Anti Spam)
- We assembled and sent a white list of known newsletters to T-Online in order to help reduce false positives. They refused the usage because we mailed them the list unencrypted over the Internet and they consider the data as disclosed.
- Tomorrow, phone conference with T-Online to discuss status.
- A white paper on how to measure spam filtering efficiency was written and sent to T-Online.

7. Extranet
- Wiebke returned from maternity period and will work 10-12 hours per week on website design.

8. Product Marketing Manager
- Two interviews conducted but no success.

9. Content Reporter
- AutoNation received a new version to solve performance problems. They conducted a larger test but did not provide any feedback so far. We believe problem is fixed, thus work on MaxDB can continue.
- Solving the MaxDB performance problems are a challenge. The progress looks promising, however the risk of a show stopper remains. Some preciser estimates expected for next week latest.

10. Competition Paper
A head-to-head competition paper Webwasher vs. Finjan was created and distributed.

–

Roland Cuny
Dipl.-Ing. (TU)

Chief Technology Officer & Co-Founder

webwasher AG - a CyberGuard Company
Vattmannstrasse 3
33100 Paderbom / Germany

Phone: +49 52 51 / 5 00 54-22
Fax: +49 52 51 / 5 00 54-11
E-mail: mailto:roland.cuny@webwasher.com
Visit us at: http://www.webwasher.com
http://www.cyberguard com

EXHIBIT 23

Towards a Testbed for Malicious Code Detection

R. Lo, P. Kerchen, R. Crawford, W. Ho, J. Crossley, G. Fink, K. Levitt, R. Olsson, and M. Archer
Division of Computer Science
University of California, Davis
Davis, CA 95616

Abstract

This paper proposes an environment for detecting many types of malicious code, including computer viruses, Trojan horses, and time/logic bombs. This malicious code testbed (MCT) is based upon both static and dynamic analysis tools developed at the University of California, Davis, which have been shown to be effective against certain types of malicious code. The testbed extends the usefulness of these tools by using them in a complementary fashion to detect more general cases of malicious code. Perhaps more importantly, the MCT allows administrators and security analysts to check a program before installation, thereby avoiding any damage a malicious program might inflict.

Keywords: Detection of Malicious Code, Static Analysis, Dynamic Analysis.

1 Introduction

In the past five years, there has been an explosion in the number of Trojan horses, time bombs, and viruses that have been found on computers. Furthermore, the ease with which one may write a virus or trapdoor is certainly cause for concern: in his Turing Award lecture, Ken Thompson demonstrated a simple trapdoor program which was quite effective in subverting the security of a UNIX system. The situation is even less encouraging in the personal computer arena: literally hundreds of computer viruses, time bombs, and Trojan horses exist for all of the major personal computers in use today.

However, there are techniques for coping with these problems. While one will never be able to distinguish a cleverly disguised virus from legitimate code, one may detect a not-so-cleverly hidden one. The same holds true for all malicious code: stopping a large percentage of destructive programs is considerably better than not stopping any. This idea forms the basis for a *malicious code testbed* (MCT) capable of detecting a large majority of current and future malicious programs. Such a testbed would allow one to examine a program to ascertain if it is suspicious. In the following section, we present a taxonomy of malicious code with examples. Following the taxonomy, we discuss many of the known methods of coping with malicious code. We then summarize the progress which has been made at UC Davis. Finally, we propose the idea of the *malicious code testbed*, which combines this previous work into a more effective system.

2 Taxonomy of Malicious Code

Computer security should insure that no unauthorized actions are carried out on a computer system. Security is violated when someone succeeds in retrieving data without authorization, destroying or altering data belonging to others, or locking up computer resources to make them unavailable. Malicious programs are those programs which cause such violations.

To categorize malicious activities, we may examine the following aspects of a malicious program [Table I].

What are the malicious actions?

A malicious program may not only directly retrieve or alter confidential information, but it may also modify the security state of the computer system so that an unauthorized person could access this information. Therefore, malicious activities refer to all activities leading to such consequences.

How do malicious programs obtain privilege?

Before any damage can be done, the malicious program must obtain the required privilege from an authorized user or from the operating system. A common way is to act as a Trojan horse, claiming to perform some useful functions, but performing others in addition or instead. A malicious program can also obtain privilege from the operating system by exploiting system bugs, taking advantage of administrative flaws, or faking authentications.

CH2981-1/91/0000/0160$01.00 © 1991 IEEE

	Malicious Actions	Obtain Privilege	Distribution Channels	Triggers
Covert channel	Disclose information	Installed by programmers		Useful information found
Worm	Exhaust resources, Any	Writer starts it, self-replication	Network	
Trojan horse	Any	User execution	Exchange of software	User execution
Virus	Infect programs, Any	User execution, self-replication	Contact with an infected system	Execution
Time/Logic Bomb	Any	Installed by programmers		Time/date, conditions satisfied
Trapdoor	Gain privilege	Holes implanted by programmers		Started by attackers
Salami Attack	Embezzlement	Installed by programmers		Execution

Table 1. Types of Malicious Code

How do malicious programs enter a system? Sometimes a malicious program is advertised as public domain software available in public bulletin boards; security may be compromised if any user copies and executes such programs on his computer. Another similar example is that of the Christmas Virus, which replicates by sending copies of itself to users and requesting them to execute the message. In cases of planned attacks, trapdoors previously implanted in the system are used by the malicious programs.

How are the malicious actions triggered? A malicious program may stay dormant for an indefinite period. It works normally until a scheduled moment or certain conditions are satisfied. For example, a malicious program which exploits covert channels may only be active when confidential information is being displayed on a terminal; at other times, it may sleep or perform some diversionary action.

3 Coping with Malicious Code

Presently, the majority of malicious code defenses are concerned with computer viruses. However, some are more broadly applicable to malicious code in general. Table 2 shows the applicability of some of these methods. One can classify these methods into two classes: *preventive* and *detective*. While prevention is important, detection is preferable since it does not rely on a program being in a "clean" state. Thus, detective approaches appear to be more generally applicable.

3.1 Program Access Control Lists

The first approach, *program access control lists* (PACL's) [5], consists of associating with each file in a system an access control list that specifies what programs can modify the file. This preventive approach has the effect of limiting damage that can be done by many malicious programs, but it is ineffective against attacks such as covert channels which only violate information security, not integrity.

3.2 Static Analyzers

From Table 2, one can see that *static analysis* [1] can be applied to a broad class of problems. By closely examining the binary or source code of a program, static analysis attempts to detect the presence of suspicious sections in that program. However, in the most general case such detection is incomputable, resulting in a need for more selective analysis techniques. Since malicious code in general can be more smoothly integrated with the code of the program it is infecting, detection must be focussed on the strategic vulnerabilities of the operating system and underlying architecture in question. In this way, more generalized detection is possible without the full cost of program verification because slicing [1] and other static and dynamic analysis tools will reduce the problem space to a tractable size.

3.3 Simple Scanners & Monitors

Simple scanners are by and large the most common means of malicious code detection in use today. Typ-

	PACL	Static Analyzer	Simple Scanner	Run-time Monitor	Encryption	Watchdog Processors	Dynamic Analyzer
Covert Channel	none	low	none	limited	high	none	high
Worm	high	low	none	low	none	none	low
Trojan Horse	high	high	low	high	low	none	high
Virus	high	high	low	high	high	high	high
Time Bomb	high	high	low	high	low	none	high
Trapdoor	none	high	none	none	low	none	high
Salami	none	low	none	none	none	none	high

Table 2. Applicability of Defenses.

ically, a scanner will search a program for patterns which match those of known malicious programs. As a result, these programs boast a very good record in defending against known malicious programs but they cannot be applied in general to finding new or mutated malicious code. Another popular approach uses *simple monitors* to observe program execution. Such monitors usually watch all disk accesses to insure that no unauthorized writes are made. Unfortunately, for these programs to be effective, they must err on the conservative side, resulting in many false alarms which require user interaction.

3.4 Encryption & Watchdog Processors

Encryption is another method of coping with the threat of malicious code. Lapid, Ahituv, and Neumann [2] use encryption to defend against Trojan horses, trapdoors, and other problems. When correctly implemented, such a system would be quite effective against many types of malicious code, but the cost of such a system is high due to the required hardware. Similarly, *watchdog processors* [3] also require additional hardware. Such processors are capable of detecting invalid reads/writes from/to memory but they would require additional support to effectively combat viruses. Also, both of these methods are preventive in that they require a "clean" version of every program which is to be examined. In many instances, such clean copies are not available, thereby limiting the usefulness of these approaches.

3.5 Dynamic Analyzers

Finally, *dynamic analysis* offers a reasonable potential for detection of a large class of malicious code. By observing a program at run-time in a controlled environment, one can determine exactly what it is trying to do. However, like static analysis, this technique must be used "off-line" to allow the analyzer to keep track of the program's actions. As a result, clever programs can elude the analyzer by only executing when they "know" that they are not being watched.

Unlike most virus detection techniques, two types of analysis attempt to peer inside a program to detect what it is doing and how. Static analysis methods can determine certain properties for some types of programs. Dynamic analysis methods attempt to learn more about a program's behavior by actually running it or by simulating its execution.

At UC Davis, three analysis tools have been developed which help in the determination of whether a program has any potentially malicious code in it: VF1, Snitch, and Dalek. VF1 uses data flow techniques to statically determine names of files which a program can access. Snitch statically examines a program for duplication of operating system services. Dalek is a debugger which forms the basis for a dynamic analyzer.

4 Static Analysis Tools

4.1 VF1

VF1 is a prototype system that has been implemented to determine the viability of applying static analysis to the detection of malicious code; it uses a technique called *slicing*. Slicing involves isolating the portions of a program related to a particular property in which one is interested. The sliced program can then be analyzed to give information about that particular property. VF1's target property is filename generation–in particular, which files can be opened and written to by a given program. By knowing what files a program can write to, one can determine if there is a possibility of the program being a virus. For example, if a program that does not need to write to files (e.g., *ls*, the UNIX directory listing program), possesses code to open and write any file, then one might be suspicious that the program contains a virus.

VF1 translates a program written in the C programming language to a program expressed in a Lisp-like intermediate form that is easier to analyze. This resultant program can then be sliced with respect to any given line of its body. That is, one can select a line of the resultant program that performs an action one is interested in (such as opening a file for writing) and VF1 will determine which statements of the resultant program have bearing on that selected line.

4.2 Snitch

Snitch is a prototype of a *detector* of duplication of operating system calls. This detector makes use of the fact that most UNIX programs contain at most one instance of any operating system service (e.g., open, write, close). Since a simple virus cannot rely on all programs possessing the services it needs, it will carry all of those services with it, inserting them into every program it infects. This will most likely result in a duplication of some OS services. When Snitch is used to analyze the infected program, it will report this duplication as being suspicious. The Snitch prototype is specific to Sun-3's running SunOS, but many of the concepts underlying the prototype can be applied to other architectures and operating systems.

Snitch consists of two major modules. The first module, the disassembler, takes an executable program as input and produces the equivalent Motorola 68020 assembly language representation as output. The second module, the analyzer, takes the output from the disassembler and examines it for duplication of OS services, reporting any such duplications as well as the number of occurrences of all system calls.

5 Debugger-based Dynamic Analysis

One obvious approach to dynamic analysis is to base the analysis on a debugger. Over the last two years, a debugger called Dalek has been developed at UC Davis [4]. Dalek offers support for the notion of user-definable *events*. The user defines an event template by writing Dalek language code (e.g., employing IF or WHILE statements) that will be executed by Dalek as it attempts to recognize different occurrences of that event. One typical form of primitive event might be defined to capture certain details of a procedure's invocations, e.g., the values of its actual parameters. Another typical form of primitive event might be defined to capture the value of a particular variable every time it changes.

Hierarchical events can also be defined. High-level events are used to correlate and combine (e.g., through Dalek's IF or WHILE statements) the attributes from instances of two or more primitive events that may have occurred widely separated in time. In this way, the user can construct behavioral abstractions - models or patterns that characterize the activity of the application program.

One can imagine how such capabilities might be applied to the detection and understanding of viruses or other malicious code but it might seem that in real-world situations, such event-based methods would be ineffective against hostile or secretive programs. In the first place, one would expect that the malicious code would have been stripped of all (correct) symbolic information. Thus the debugger would not know the names, sizes, or locations of procedures or data structures. However, most operating systems offer some assistance in this regard, allowing a relatively complete behavioral trace of all system-related activity initiated by a suspicious program to be obtained. Secondly, a malicious program may alter its own code, making analysis difficult. Under Dalek, however, one may define events to recognize such self-modifying behavior. Therefore, self-modification does not present insurmountable difficulties for the debugger but it does increase its complexity.

Figure 1 illustrates how high-level events can be used to correlate attributes captured by lower-level events to provide a characterization of a suspicious program's behavior represented in terms of whatever semantic models the user has determined are most relevant.

We envisage equipping Dalek with a library of predefined events to capture suspicious and malicious behavior, similar in spirit to the events shown in Figure 1. For example, attempting to open (or change/inspect the permissions on) all files in the current directory might be considered suspicious. Writing the same block of "data" to several different executable files would appear even more suspicious.

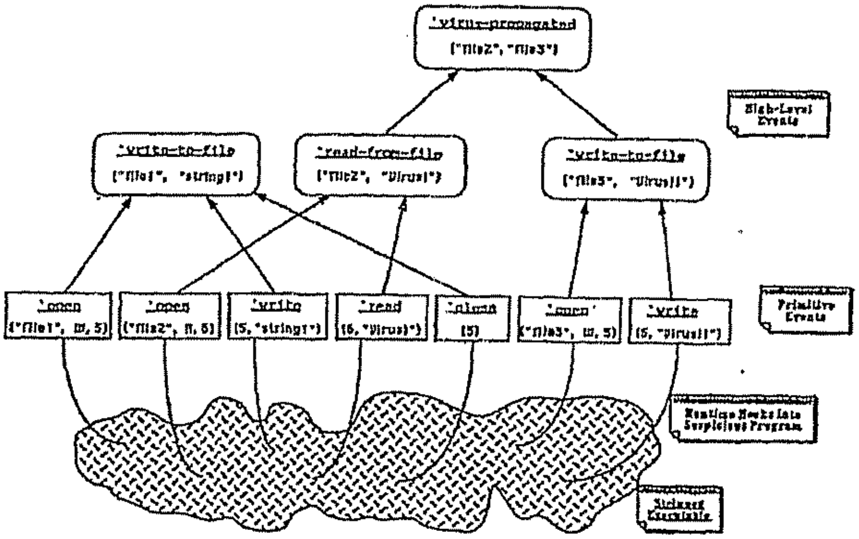


Figure 1. Interaction of Events in Dalek

6 Towards a Testbed

The *malicious code testbed* (MCT) under development consists of a set of tools that will assist a user in detecting viruses and Trojan horses and in identifying programs which exploit security flaws within developed software. It is based in part on the three tools mentioned above: Dalek, VFI, and Snitch.

The primary goal is to provide an environment and tools to assist in the identification of malicious logic in developed software. Since malicious code detection is an incomputable problem, the tools will not be able to give a yes-no answer. Instead, the software is analyzed and its properties summarized to allow the analyst to understand the effect of its execution. The tools will identify suspicious code but it is up to the user to make the final decision about whether or not the code is malicious. For example, our tool may indicate that a program would destroy all information in the current directory. Most people would consider this a malicious activity. However, the program is not malicious if the intention of the user is to clean up his directory by using such a program.

The other goal is to further examine a suspicious program identified by the MCT. The purpose of this further examination is to determine the severity of the identified suspicious activity, locate other suspicious activities, determine its triggering conditions, and produce signatures that may be used to locate the existence of identical or similar malicious logic in other programs.

The MCT employs two kinds of analysis techniques: *static analysis* and *run-time*, or *dynamic*, *analysis*. Both techniques are necessary because they are applied in different situations, thus complementing each other. Compared with static analysis, dynamic analysis is less computation intensive and able to follow any execution sequence even if the program modifies itself on the fly. However, since only some executed sequences are tested, dynamic analysis can certify only the existence of certain activities, i.e. violation of security policy, but it cannot indicate their non-existence. Therefore, both are needed.

Architecture

The static analysis tool works in 5 stages: processor-dependent disassembly, intelligent decompilation, data flow analysis, slicing, and symbolic simplification. The processor-dependent disassembly stage translates an executable program into an intermediate form, and then the decompiler attempts to determine how variables are allocated in the program. Knowing where the variables are stored, the data flow analyzer determines the relationship between variables, i.e. which variables influence the value stored in designated variables. Slicing produces a bona-fide program that computes the value of the variables in question. Finally, the symbolic simplifier tries to simplify the bona-fide program as much as possible.

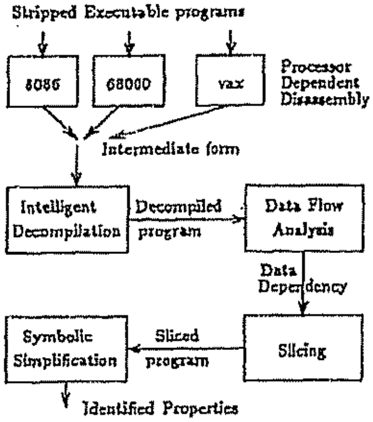


Figure 2. Architecture of the static analyzer.

6.1 Complementary Use of Static and Dynamic Analysis

After some degree of intelligent decompilation, the static analyzer attempts to slice the program to identify all sections of the code involved in the generation of filenames.

Static analysis also needs to look for other discernible suspicious properties. Suppose, for example, that the code resultant from slicing with respect to filename generation is not reachable from the main entry point of the original program. This indicates a very poorly written program or a program that modified itself in order to reach the sliced section. Similarly, any other indications of self-modifying behavior would be grounds for more extensive dynamic analysis.

Dynamic analysis can force the suspicious program to execute certain sections, suspending it periodically to communicate its status to the static analyzer. During this phase, the user of the MCT might devise various hypotheses explaining the goals of the suspicious program and explaining its methods in pursuit of those objectives. For example, a program which encrypts part of its code would be analyzed dynamically in order to examine the decrypted code. This iterative process could continue until the MCT user had gained a thorough understanding of the goals and methods of the suspicious program.

An Example

The ftp program associated with Sun Unix 3.4 has a bug that allows a masquerader to login as any other user provided that he has successfully logged into the system once. The masquerader first logs in with a valid userid and password, causing the *logged_in* variable to be set to 1. Then s/he performs a login with a 'victim' userid and any password, exploiting the flaw that the *logged_in* variable is not reset. Since *check_login* only checks *logged_in*, the 'victim' userid is assumed to be logged in the system. The essential stripped code, written in pseudocode, is as follows:

```
ftp:  USER username CR
          { set new user id
                ... no reset of the variable logged_in }
      | PASS password CR
          { if password is correct, set logged_in to 1
      else print error message }
check_login:
      { valid_login == logged_in; }
```

This bug can be identified with static techniques in two ways. The first method is to check for a data flow anomaly in the data dependency graph. We can see that the variable *logged_in* is never initialized in the program. The second method is to expand *logged_in* symbolically to see how it is computed. The symbolic output will indicate that its value is set with a correct password, but not changed with an incorrect password.

7 Conclusions and Future Work

We have described a testbed under development which detects malicious code that other techniques cannot detect. This testbed uses static and dynamic analysis techniques in a complementary fashion to identify suspicious programs before they are installed and allowed to cause any damage. The static analysis uses slicing to reduce a program to a size which allows verification techniques to discover any suspicious code. The dynamic analysis uses an event-based debugger which is capable of analyzing code which static analysis cannot. We will be applying this testbed on known instances of malicious code, especially viruses, worms, and the like. Future work will include formalizing the concepts of *maliciousness* and *suspiciousness* and improving the static analysis techniques used to discover the meanings of loops in sliced programs.

Acknowledgements

We thank Doug Mansur and Bill Arbaugh for their valuable insights.

References

[1] P. Kerchen, R. Lo, J. Crossley, G. Elkinbard, K. Levitt, and R. Olsson. "Static Analysis Virus Detection Tools for UNIX Systems", *Proc. of NIST/NCSC 13th Nat'l Computer Security Conf.*, Washington, DC, Oct. 1-4, 1990, pp. 350-365.

[2] Y. Lapid, N. Ahituv, and S. Neumann. "Approaches to Handling 'Trojan Horse' Threats", *Computers & Security*, Vol. 5, 1986, pp. 251-256.

[3] A. Mahmood and E. J. McCluskey. "Concurrent Error Detection Using Watchdog Processors–A Survey" *IEEE Transactions on Computers*, Vol. 37, No. 2, 1988, pp. 160-174.

[4] R. Olsson, R. Crawford, and W. Ho. "A Dataflow Approach to Event-based Debugging", to appear in *SOFTWARE-Practice and Experience*.

[5] D. Wichers, D. Cook, R. Olsson, J. Crossley, P. Kerchen, K. Levitt, and R. Lo. "PACL's: An Access Control List Approach to Anti-Viral Security", *Proc. of NIST/NCSC 13th Nat'l Computer Security Conf.*, Washington, DC, Oct. 1-4, 1990, pp. 340-349.